View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Technische Universität Dresden: Qucosa

Der Fakultät Mathematik und Naturwissenschaften
der Technischen Universität Dresden

Solving multi-physics problems using adaptive finite elements with independently refined meshes

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum naturalium
(Dr. rer. nat.)

von
Siqi Ling

geboren am 18. September 1987
in Shanghai, China

Tag der Einreichung:
Tag der Verteidigung:

Gutachter: Prof. Dr. rer. nat. Axel Voigt
Technische Universität Dresden
Prof. Dr.-Ing. Jeronimo Castrillon
Technische Universität Dresden

Abstract

In this thesis, we study a numerical tool named multi-mesh method within the framework of the adaptive finite element method. The aim of this method is to minimize the size of the linear system to get the optimal performance of simulations. Multi-mesh methods are typically used in multi-physics problems, where more than one component is involved in the system. During the discretization of the weak formulation of partial differential equations, a finite-dimensional space associated with an independently refined mesh is assigned to each component respectively. The usage of independently refined meshes leads less degrees of freedom from a global point of view.

To our best knowledge, the first multi-mesh method was presented at the beginning of the 21st Century. Similar techniques were announced by different mathematics researchers afterwards. But, due to some common restrictions, this method is not widely used in the field of numerical simulations. On one hand, only the case of two-mesh is taken into scientists' consideration. But more than two components are common in multi-physics problems. Each is, in principle, allowed to be defined on an independent mesh. Besides that, the multi-mesh methods presented so far omit the possibility that coefficient function spaces live on the different meshes from the trial and test function spaces. As a ubiquitous numerical tool, the multi-mesh method should comprise the above circumstances. On the other hand, users are accustomed to improving the performance by taking the advantage of parallel resources rather than running simulations with the multi-mesh approach on one single processor, so it would be a pity if such an efficient method was only available in sequential. The multi-mesh method is actually used within local assembling process, which should not be conflict with parallelization. In this thesis, we present a general multi-mesh method without the limitation of the number of meshes used in the system, and it can be applied to parallel environments as well. Chapter 1 introduces the background knowledge of the adaptive finite element method and the pioneering work, on which this thesis is based. Then, the main idea of

the multi-mesh method is formally derived and the detailed implementation is discussed in Chapter 2 and 3. In Chapter 4, applications, e.g. the multi-phase flow problem and the dendritic growth, are shown to prove that our method is superior in contrast to the standard single-mesh finite element method in terms of performance, while accuracy is not reduced.

Kurzfassung

Diese Arbeit beschäftigt sich mit der Multi-Mesh-Methode, die auf dem Gebiet der adaptiven Finiten-Elemente-Methode Anwendung findet. Das Ziel der Multi-Mesh-Methode ist es, die Größe des durch die Assemblierung entstehenden linearen Gleichungssystems zu verringern, um damit der eine optimale Performance zu erreichen. Diese Methode wird insbesondere in multiphysikalischen Problemen eingesetzt, die aus mehr als einer Komponente bestehen. Durch sie wird jeder einzelnen Komponente bei der Diskretisierung der schwachen Form der partiellen Differentialgleichungen ein endlichdimensionaler Raum mit einem unabhängig verfeinerten Gitter zugeordnet. Dadurch verringern sich die Freiheitsgrade der Diskretisierung. Nach unserer Kenntnis wurde eine erste Multi-Mesh-Methode Anfang des 21. Jahrhunderts vorgestellt und durch ähnlich Methoden darauffolgend ergänzt. Jedoch konnte sich die Multi-Mesh-Methode nie ganz durchsetzen. Einerseits verwendete man nur Two-Mesh-Techniken, was jedoch bei multiphysikalischen Problemen mit mehreren Komponenten eine starke Einschränkung bedeutet. Anderseits erlaubten bisherige Multi-Mesh-Methoden nicht, dass die Räume der Koeffizientenfunktionen auf anderen Gittern leben als die Räume der Ansatz- und Testfunktionen. Als universell anwendbares numerisches Werkzeug sollte auch die Multi-Mesh-Methode solche Fälle berücksichtigen. Weiterhin ist es unabdingbar für das moderne wissenschaftliche Rechnen, die Methode auf parallele Berechnungsverfahren zu erweitern, um damit eine weitere Performancesteigerung zu gewährleisten. Um den Forderungen Rechnung zu tragen, erweitern wir in dieser Arbeit das bisherige Verfahren und beschreiben damit eine allgemeine Multi-Mesh-Methode zur Verwendung beliebig viele Gitter, die auch parallele Berechnungen erlaubt. In Kapitel 1 erläutern wir die wichtigsten Grundlagen für dieses Gebiet und beschreiben außerdem die Pionierarbeit, die hierfür geleistet wurde. Die Grundidee dieser Methode wird hergeleitet und detailliert implementiert in den Kapiteln 2 und 3. Schließlich beschreiben wir im Kapitel 4 die Anwendung der Methode auf Mehrphasenströmungen und Dendritenwachstum. Dabei

zeigen wir, dass die Multi-Mesh-Methode einfachen Single-Mesh-Verfahren in Bezug auf die Effizienz weit überlegen ist, ohne die Genauigkeit des Verfahrens zu beeinträchtigen.

Acknowledgement

Three years have past since I became a member of the Institute of Scientific Computing at the Technische Universität Dresden. I was not original in this field and it was a great challenge for me. Fortunately, everyone in the institute is very nice and kind. I could get help whenever I asked for. I really had a wonderful time here and now I am one step away from the end. I am not the one who likes saying goodbye but when the time comes, everyone has to move on.

I would like to express my sincere thanks to my supervisor, Prof. Dr. rer. nat. Axel Voigt, who offered me this valuable opportunity to work in such a nice group. It is because of his help that I managed to customize the topic and the idea of my research field, otherwise these three years would be much more difficult. Besides my supervisor, my colleagues also supported me throughout my work and I would like to appreciate them. Thank to Dr. Simon Praetorius who introduced me to the theory of the finite element method and our toolbox AMDiS. I have asked numerical questions to him within the past three years and he always replied me with patience. Thank to Dr. Wieland Marth for his multi-phase flow problem, which is a very important section in this thesis. It was a great pleasure to collaborate with him. Furthermore, I would like to thank Dipl.-Math. Andreas Naumann, Dr. Sebastian Aland, Dipl.-Math. Matthias Wagner, M.Sc. Francesco Alaimo and those whoever helped me. I also want to thank the Center for Advancing Electronics Dresden (cfaed) for the three-year funding, especially the nice co-workers from the Orchestration Path. Moreover, I would like to thank all those who helped me in proofreading this thesis. Last but not least, thank to my family for their understanding and support of my study in Dresden.

Contents

CHAPTER 1

Introduction

Scientific computing is a rapidly growing multi-disciplinary field that solves complex physical problems by exploiting advanced computing capabilities. Various mathematical methods are used within this field. One of the most prominent challenges in scientific computing is the solution of partial differential equations (PDEs). For this task, the finite element method (FEM) is proved to be one of the most integrated and popular numerical tools. The theme of this thesis is about a numerical approach named multi-mesh method, which is an advanced sub-method used in the assembling process of the finite element method.

This chapter is divided into three parts. First, in Section 1.1, we give a brief overview on the finite element method, which includes the history of FEM, the solution of an example PDE using FEM, and some more discussion on adaptivity and error estimation theory, which are very important features of today's FEM. From Section 1.2 we go deeper into the implementation. Different kinds of finite element toolboxes are mentioned, but we mainly focus on our library: **A**daptive **M**ulti**Di**mensional **S**imulations (AMDiS), including its basic concepts and data structures. Although the idea presented in this thesis is not restricted to any specific finite element software, we give explanations by means of the terminology and the notations used in AMDiS since all the work is implemented within the AMDiS framework. Moreover, the "software concepts" sections in Chapter 2 and 3 are directly related to the implementation details. In Section 1.3, we return to our subject, the multi-mesh method, its history and variants. The pioneer work, on which this thesis is based, is also introduced.

1.1 Finite element method

1.1.1 History

When we talk about the history of the finite element method, it is hard to target the precise father and the birthday of this method, but we do distinguish several pioneers who have contributed to the invention of FEM. In the early 1940s, the first FEM-style calculation on a triangular net for the piecewise linear approximation of the stress function in the torsion problem was presented in R.Courant's paper. In 1950s, M.J.Turner et al. at Boeing generalized and perfected the direct stiffness method where the triangular element stiffness matrix was introduced. Inspired by Turner's work, J.H.Argyris was the first in constructing a displacement-assumed continuum element. R.W.Clough continued convergence studies on stress components and contributed to popularize the ideas by giving the name "finite element method". O.C.Zienkiewicz clarified and further systematically developed the potential energy minimisation theory from R.Courant and published the first text book about the finite element method in 1957. All these mentioned scientists [34] are largely responsible for the popularization of FEM from aircraft structural engineering to a wider range of new application fields such as metal forming, electromagnetics, geomechanics, biology, etc.

One common way to get a more precise approximation is to increase the number of elements on the mesh. Smaller elements help to minimize the discretization error, but on the other hand, overall simulation time increases as a result of more computing elements. In order to keep a balance between accuracy and efficiency, the adaptive method was introduced. It was first addressed by Babuška and Rheinboldt in 1970s [2]. They showed the possibilities of economic error estimation and indicated a desired accuracy of the numerical solution could be reached by subdivion of meshes. Later in the 1980s and 1990s, a great deal of effort was contributed to the design of adaptive method and error estimation theory, following the pioneering work of Babuška. This subject became a widely popular research area, in which adaptive methods such as h-refinement (changing the number of elements on the mesh), p-refinement (changing the degree of interpolation functions) and h-p combinations, error estimation procedures such as a priori and a posteriori error estimators, were investigated. In 1980s and 1990s, a booming activity in the design of different kinds of a posteriori error estimators was fostered. Today, a posteriori error estimators are well developed for a large class of simple linear elliptic model problems. For more details on the history of finite element methods, we refer to JL.Meek [34] and O.C.Zienkiewicz [54].

When the finite element method first came out, computers were so expensive that only large industrial companies could afford them. But with the growing accessibility of computational resources as well as the perfection and the popularization of FEM, it has become a well-developed numerical instrument in both science and engineering for

modelling natural systems in physics, chemistry, biology and so on.

1.1.2 Finite element discretization

The finite element method discretizes the components of PDEs by finite element spaces. A finite element space $\mathcal{V}$ has three factors. First, a geometry domain Ω consisting of finite elements. Second, a set of global basis functions denoted as $\chi = \{\chi_1, ..., \chi_n\}$, which is used to approximate the true solution. Third, a mapping from element-wise local basis functions to global basis functions. In order to illustrate the discretization via the finite element space $\mathcal{V}$, we consider the following second order differential equation with Dirichlet boundary conditions as an example. The equation reads as:

$$-\nabla \cdot A\nabla u + b \cdot \nabla u + cu = f \qquad \text{in } \Omega \tag{1.1a}$$

$$u = 0 \qquad \text{on } \partial\Omega \tag{1.1b}$$

, where $A \in L^\infty(\Omega; \mathbb{R}^{d\times d})$, $b \in L^\infty(\Omega; \mathbb{R}^d)$, $c \in L^\infty(\Omega)$ and $f \in L^2(\Omega)$. The same kind of equations result from a linearization of nonlinear elliptic problems. Note that in the AMDiS environment, each part of the equation split by a plus operator is called an "operator term". Thus, $\nabla \cdot A\nabla u$, $b \cdot \nabla u$ and cu are called second, first and zero order terms respectively. To get a variational formulation (weak formulation), we multiply both sides of the PDE by a function v, which is called test function, such that $v = 0$ on $\partial\Omega$. After integration by parts, we have the weak formulation of the original equation as:

$$\int_\Omega f(x)v(x)dx = \int_\Omega \nabla\ v(x) \cdot A(x)\nabla u(x) + v(x)b(x) \cdot \nabla u(x) + c(x)v(x)u(x)\ dx \tag{1.2}$$

Then, we partition the domain Ω to a set of finite elements $S = \{s_1, ..., s_m\}$, where m is the total element number. We denote our finite element space of globally continuous, piecewise polynomial functions P^n as $\mathcal{V} := \{u_h \in H_0^1 : u_h|_s \in P^n\ \forall s \in S\}$, where n is the polynomial degree. We thus obtain: find $u_h \in \mathcal{V}$ such that:

$$\int_\Omega f(x)v(x)dx = \int_\Omega \nabla\ v(x) \cdot A(x)\nabla u_h(x) + v(x)b(x) \cdot \nabla u_h(x) + c(x)v(x)u_h(x)\ dx \tag{1.3}$$

Furthermore, we define $\chi = \{\chi_1, ..., \chi_n\}$ to be the basis of $\mathcal{V}$. We can replace u_h by the linear combination $u_h = \sum_j^n u_j\chi_j$ with u_j the unknown real coefficients. Using these relations and breaking up the domain in the partitions of Ω, we rewrite Eq. (1.3) as:

$$\sum_{s\in S}\int_s f\chi_i = \sum_{j=1}^{n} u_j \left(\sum_{s\in S}\int_s \nabla\ \chi_i \cdot A\nabla\chi_j\right) + \sum_{j=1}^{n} u_j \left(\sum_{s\in S}\int_s \chi_i b\cdot\nabla\chi_j\right) + \sum_{j=1}^{n} u_j \left(\sum_{s\in S}\int_s c\chi_i\chi_j\right) \qquad \forall\chi_i \in \chi \tag{1.4}$$

The whole n equations can be written in the form of a matrix. The resulting matrix is called global stiffness matrix. Finally, we can obtain the approximate solutions after applying appropriate solver methods.

1.1.3 Adaptive method

Today, most finite element codes use adaptively refined meshes. Adaptivity has become one of the key attributes of an efficient finite element method. In this section, we focus on adaptivity and error estimation theory. We start with a sketch of the algorithm of adaptive finite element methods (AFEMs), shown in Fig 1.1. At the starting point, the algorithm requires three input data: a system of PDEs associated with boundary conditions, predefined basis functions, and a mesh representing the geometry. The main body of the algorithm is a loop named "adaption loop", which includes five components: assembler, solver, estimator, marker and adaptivity [35]. The interesting component in terms of error estimation theory is the error estimator, which serves as an indicator where the difference between the approximate solution and the exact solution caused from discretization in a specified norm is relatively large. It computes the global error by collecting the local error on each element. Basically, there are two types of error estimation procedures: a priori error estimators and a posteriori error estimators. For review, we refer to T.Grätsch [20]. The first a posteriori error estimator we use is the residual error estimator. we adopted the indicator described by K.Eriksson and C.Johnson [16] and R.Verfürth [47]. Depending on the finite element solution u_h, the form reads as:

$$\eta(u_h) = \sum_{s\in S}\left(c_1 R_s(u_h) + c_2 \sum_{e\subset\partial s} J_e(u_h)\right) \tag{1.5}$$

where R_s is the residual on element s and J_e the jump residual defined on the edge e of element s. c_1 and c_2 are the constants corresponding to the element residual and the jump terms. Another a posteriori error estimator we use is the recovery estimator, which is based on the recovery gradient of the solution. Note that the gradient of the finite element solution is in general discontinuous across the inter-element boundaries. The recovery gradient is a smoothed continuous version of this gradient [48].

In the case of a relatively large overall error, the fourth component, the marker, is used to identify the parts of the mesh, where the estimated error should be decreased,

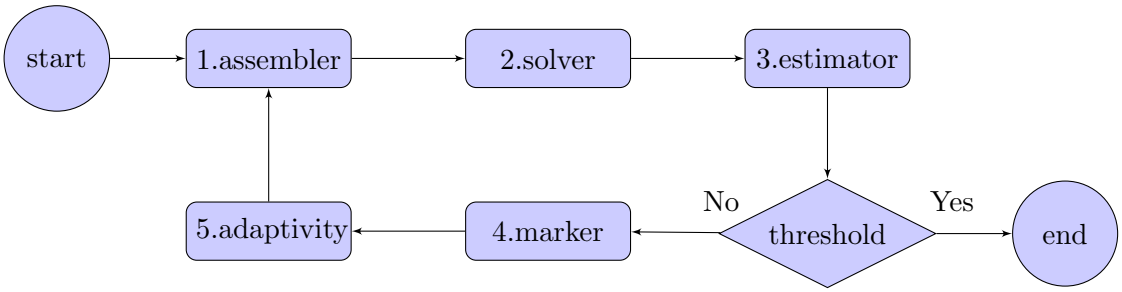


Figure 1.1: Sketch of adaptive finite element methods

according to the result of the estimator. Global refinement, maximum strategy, equidistribution strategy and guaranteed error reduction strategy are the four h-refinement marking strategies implemented in AMDiS. Then, adaptivity follows directly after the marking, performing refinement or coarsening on the mesh.

In principle, it is possible to adopt different strategies for different components. Even within one component, strategies are allowed to be combined. For instance, a marker marks an element to be refined if at least one strategy has marked the element to be refined. On the contrary, an element is coarsened if all strategies have marked it to be coarsened.

After the adaptivity, the adaption loop goes back to the assembling and a new iteration begins. The loop stops only if the global error is under the threshold (a predefined tolerance) or the number of iterations exceeds the given maximum value.

1.2 AMDiS

1.2.1 FEM toolboxes

Today, people can find hundreds of finite element packages and toolboxes on the Internet. The specification of those toolboxes varies from one to another. Each has its own advantages and drawbacks. Some of the packages are designed for general problems, others are highly specialized for some specific partial differential equations. They are also implemented in different kinds of languages, commonly in Matlab, Fortran and C/C++, etc. Here, we give a link of the commercial and open-source FEM packages accumulated by wikipedia: `https://en.wikipedia.org/wiki/List_of_finite_element_software_packages`. In this thesis, we are working with AMDiS. It is an open source finite element library developed in the institute of scientific computing, TU Dresden, which supports the solution of a large class of stationary and instationary systems of PDEs.

Within the last 15 years, many developers have already put their effort into making AMDiS both user-friendly and efficient by means of well-designed abstract data structures and modern software concepts. AMDiS supports operator splitting, problem coupling, time discretization schemes (implicit, semi-implicit, explicit Euler scheme), boundary conditions (Dirichlet, Neumann, periodic, etc) and so on. Further discretization schemes, e.g. mixed-element, Taylor-Hood method, multi-mesh method, are also at users' hands. In terms of solution methods, we did not restrict ourselves to one specific linear solver, but instead, we presented a framework that allows the implementation of a large class of direct and iterative solver methods with standard and problem-based preconditionings.

Some of the AMDiS software concepts are initially derived from ALBERTA [40], the most important of which are introduced in the next section. Nowadays, AMDiS goes far beyond ALBERTA. It supports the distribution of geometric information and a broad range of parallel solvers, based on distributed memory systems. Nice weak and strong scalability are shown for up to 16.000 processors (so far we have tested).

Due to high level of abstraction in AMDiS, users do not need to pay attention to the details of the adaption loop and all the inner components. They can start simulations just by a call to the function adapt. But beforehand they have to write their partial differential equations, provide a specific domain, and choose the strategies, which are suitable for their applications. In the background, AMDiS takes the usage of the available hardware resources and performs all the computing for users. See the AMDiS Tutorial [49] for more information on how to solve PDEs with AMDiS.

1.2.2 Basic concepts in AMDiS

For the readers of this thesis, it is better to have a fundamental knowledge about the implementation, which is introduced in this section. We try to answer the questions such as "What kind of elements are used in AMDiS?" and "How does a mesh look like in the storage of memory?", etc.

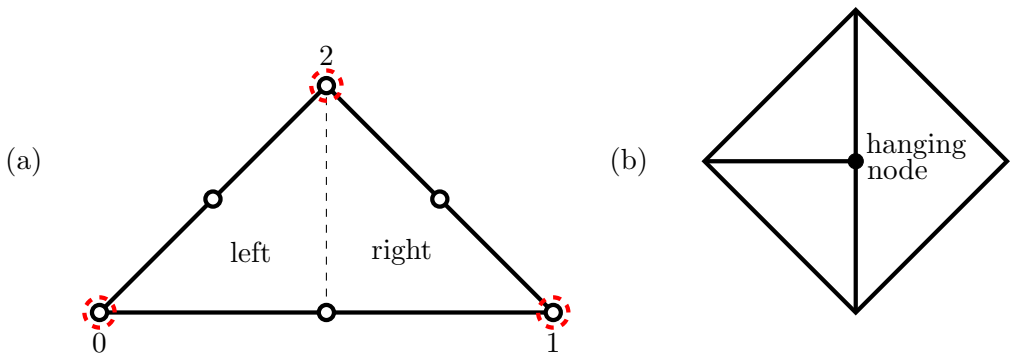
1.2.2.1 Elements and meshes

Like in other FEM toolboxes, an AMDiS mesh is formed by the union of elements. Those elements are simplices (triangles in 2D and tetrahedrons in 3D) and any two of them are either disjoint or share a common boundary. The coarsest unrefined mesh is called a "macro mesh". It consists of macro elements that are provided in a geometry information file named "macro file". In Fig 1.2(c), a 2D macro mesh of a squared domain is shown, which is built of four macro triangles. Note that we only allow conforming meshes, i.e. there is no hanging node on meshes. Fig 1.2(b) shows a hanging node in 2D. One of the neighbor triangles sharing a common edge is refined, while another is not.

Then, the midpoint on the refinement edge, which does not properly belong to both of the triangles, is the hanging node. The refinement algorithm we use is responsible for keeping meshes conforming.

The refinement strategy in AMDiS is the bisection algorithm. When one element is marked to be refined, the longest edge is marked as the refinement edge. Then, the element is split into a left-child and a right-child by cutting the refinement edge at its midpoint. For the simplicity to distinguish the refinement edge, the vertices of the element are enumerated in a fixed sequence. In our convention, the left vertex of the longest edge is given index 0, and the remaining vertices are numbered counter clockwise, see Fig 1.2(a). In 3D, the enumeration of vertices depends on the type of the tetrahedron [40]. The index of the newly generated vertex at the midpoint of the refinement edge has the highest local index within both children elements. Note that children elements are no longer macro elements. Typically, if more than one mesh is used, those meshes come from the same macro mesh, but with a different refinement set.

After refinement is performed on a macro element recursively, a refinement hierarchy is created and stored in the form of a binary tree, recording the information whether it is a left- or right-child for each refinement level. Then, the refinement hierarchy of the whole mesh is represented by a bunch of binary trees. The binary tree of macro element 0 in Fig 1.2(c) is shown in (d). The refinement hierarchy is one of the most important information, which is frequently in use, for example, for mesh repartitioning and status recovery. Thus, we need a more compact format than a binary tree. The format we developed is called "Mesh Structure Code". The origin of this concept comes from [48]. The idea is that we traverse the binary tree using pre-order traversal, and for each refinement level, we store a 1 if the corresponding element is refined and a 0 if not. The resulting binary sequence can be interpreted as one (unsigned long) integer value that can be sent across processors efficiently. If the integer capacity is reached, then an array of integer is used. The lower part of Fig 1.2(d) shows the mesh structure code of the binary tree of macro element 0.



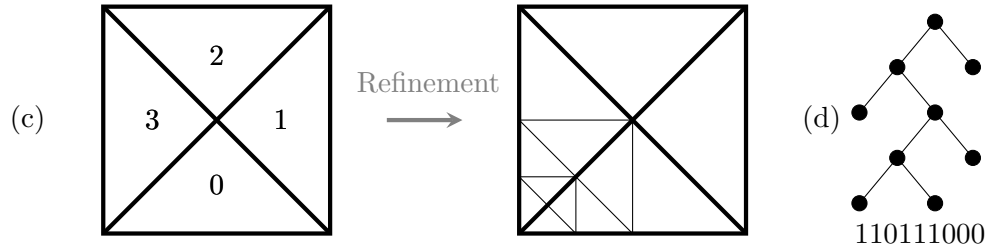


Figure 1.2: (a) Local vertex indices of a triangle is shown. A linear (dashed red) and a Quadratic (black) Lagrange basis functions are defined on the triangle. The potential bisection of this triangle is also shown. (b) a hanging node in 2D (c) a macro mesh consisting of four macro elements before and after mesh refinement (d) the refinement hierarchy of macro element 0 in (c) and the corresponding mesh structure code

1.2.2.2 Degrees of freedom

Instead of explicit storage, global basis functions are commonly constructed from the sum of all local basis functions that share the corresponding global indices. So, the third factor of the finite element space $\mathcal{V}$, the mapping from local to global basis functions, is used, which is denoted as $G_s^{\mathcal{V}}(i) \rightarrow \{1, ..., n\}, i = 1, ..., n_B$, where n is the total number of global basis functions and n_B depends on the dimension and the function we choose. The mapping returns the corresponding global index of local index i, located on element s. In AMDiS, the mapping $G_s^{\mathcal{V}}(i)$ is done by storing the global indices at the element nodes, which are named degrees of freedom (DOFs). A DOF can be located at vertices, edges, faces or in the center of an element. When different polynomial degrees are used for different components, multiple sets of DOFs are used. Fig 1.2(a) shows a triangle with two sets of DOFs for a linear degree (dashed) and a quadratic degree (solid) Lagrange basis functions, which is typically used in the mixed finite element method of the Navier-Stokes problem.

"DOF matrix" and "DOF vector" are the components of linear systems, whose indices are degrees of freedom as the names imply. A DOF matrix represents a global stiffness matrix and a DOF vector represents a vector of coefficient related to a special function basis, e.g. the variables in PDEs, whose approximate solutions are built from the stored coefficients. As it is possible to share one mesh between different finite element spaces, it is also possible to share one element space between multiple DOF matrices and vectors.

1.2.2.3 Summary

We have given a brief introduction about some basic concepts used in AMDiS. The relationship of these concepts is apparent from the AMDiS standard output file: AMDiS-Refinement-Hierarchy (ARH), see Table 1.2. As long as we have those information, we are able to recover simulation status from the last breakpoint. The capability of serialization and deserialization brings us convenience in handling large-scale adaptive simulations, e.g. the dendritic growth problem discussed in 4.2.

Field	**Description**
▷ FOR Macro[i], $i = 0,\ldots,$#el-1	#el: number of macro elements
#bits	number of structure code bits
code-data	structure code
▷ FOR fe[j], $j = 0,\ldots,$#fes-1	#fes: number of finite element spaces
#val	number of values per vector
▷ FOR $k = 0,\ldots,$#vec-1	#vec: number of vectors
$\text{vec}_j^k[p]$	vec_j^k: the kth vector of fe[j] , $p = 0,\ldots,$#val-1

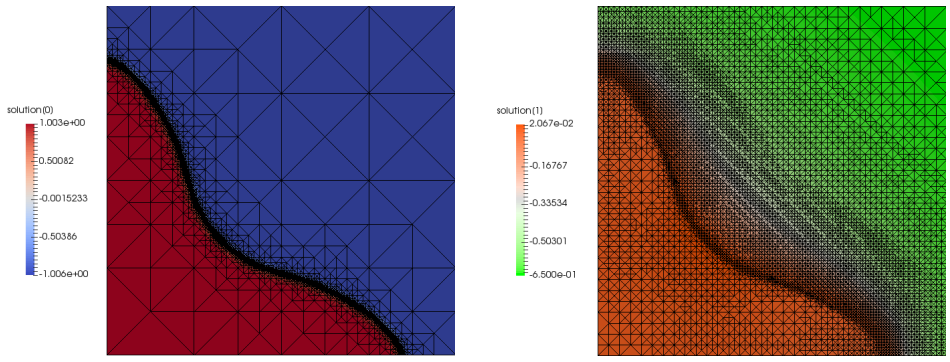
Table 1.2: Basic concepts in the storage of ARH file

1.3 Multi-mesh concepts

1.3.1 Motivation and history

In multi-physics problems, multiple physical components are involved in the system, e.g. velocity and pressure in the fluid dynamics problem, phase and thermal in the dendritic growth problem, etc. On one hand, the inner connection between the components is represented by the operator terms in the equation, which couple these components together (called "coupling term" in this thesis). On the other hand, the components behave independently with each other within the domain due to their intrinsic properties. We take the dendritic growth problem as an example to illustrate the potential difference of the behavior between components. The model we use is a phase field model. The two components in the system are a phase field variable denoted as ϕ and a thermal field variable denoted as u. In Fig 1.3, the comparison between the meshes used for the

two components separately is shown. Here, it is enough for us to know that there is a huge difference in terms of the refinement hierarchy. The reason of the difference will be discussed in Section 4.2. In this case, the application of the standard single-mesh finite element method might not be appropriate. For each component, the corresponding mesh also contains the refinement hierarchies from other components. As a consequence, the size of the final linear system increases, and so as the computation time.



ϕ u

Figure 1.3: Different behavior of the phase field variable ϕ and the thermal field variable u

To solve this problem, the multi-mesh method was developed. The main idea is to use independently refined meshes for each component. The resulting linear system is usually much smaller, when compared to the system in the single-mesh case. Thus, the overall computation time is largely saved. The usage of multiple, independently refined meshes to discretize different components in the system of PDEs is not new. To our best knowledge, A.Schmidt [39] was the first one who considers this method for the adaptive solution of coupled systems. R.Li et al. [10] also presented a similar method originally used for solving optimal control problems. Within their implementation, different meshes are a different subset of the "Hierarchy Geometry Tree": a tree data structure representing the deepest global-refined mesh. P.Solin et al. [42, 44] applied a multi-mesh method in linear thermoelasticity problems and also transient heat and moisture transfer problems. Later, the multi-mesh method was proved to be useful in more multi-physics applications [21, 27, 28, 43].

Although the multi-mesh technique is introduced, in none of these publications the method is formally derived. Furthermore, implementation issues are not discussed and detailed runtime comparison between the single- and the multi-mesh methods is missing. In contrast, T.Witkowski [51] presented a multi-mesh method in 2012, in which both the theory and the implementation of the multi-mesh method are described in detail. The

work presented in this thesis is based mainly on his work. So a short introduction of the multi-mesh approach of T.Witkowski is given in the following section.

1.3.2 Idea of the multi-mesh method

As discussed, two components, e.g. ϕ and u, are assigned to two meshes. We denote $S^{'}$ to be the mesh for ϕ and $S^{''}$ the mesh for u. Note that the usage of different meshes also means the usage of different finite element spaces. We further denote $\chi_{\phi}^{s^{'}}$ to be the local basis functions for ϕ and $\chi_{u}^{s^{''}}$ for u. The local integrals to be evaluated on each element resulting from second, first and zero order terms are:

$$\int_{s^{'}\in S^{'},s^{''}\in S^{''}} \nabla\chi_{\phi}^{s^{'}} \cdot A\nabla\chi_{u}^{s^{''}} dx \ , \ \int_{s^{'}\in S^{'},s^{''}\in S^{''}} \chi_{\phi}^{s^{'}} b \cdot \nabla\chi_{u}^{s^{''}} dx \text{ and } \int_{s^{'}\in S^{'},s^{''}\in S^{''}} \chi_{\phi}^{s^{'}} c\chi_{u}^{s^{''}} dx$$

There is no straightforward method to calculate the integrals since they now live on a pair of elements from two meshes. The element pair from mesh $S^{'}$ and $S^{''}$ is denoted as $\{s^{'}, s^{''}\}$ where $s^{'} \in S^{'}$ and $s^{''} \in S^{''}$. In order to solve this problem, we first need to make an assumption. We assume that any element $s^{'} \in S^{'}$ is either a sub-element of an element $s^{''} \in S^{''}$, or vice versa. This is not a restrict precondition since it is always fulfilled if standard refinement algorithms such as the bisection or the red-green refinement are performed on the same macro mesh. Under this assumption, our idea is that we always evaluate the integrals on the smaller element and replace the local basis functions of the larger element by a linear combination of the basis functions of the smaller one. For example, if $s^{'}$ is smaller than $s^{''}$, the zero order integral is replaced by:

$$\int_{\{s^{'},s^{''}\}} \chi_{\phi}^{s^{'}} c\chi_{u}^{s^{''}} = \int_{s^{'}} \chi_{\phi}^{s^{'}} c(\sum_{i} c_i \chi_{\phi}^{s^{'}}) \tag{1.6}$$

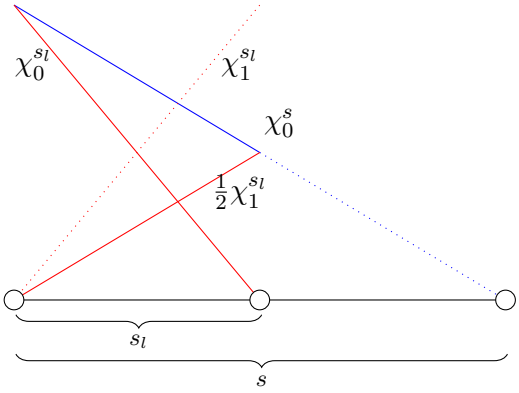
, where c_i is the coefficient. This is possible if we discretize ϕ and u by polynomial functions of the same degree. Fig 1.4(a) shows χ_0^s, the basis function on index 0 of element s, is substituted by the linear combination of the local basis functions of its left child s_l. In the case of indirect child-parent relationship between $s^{'}$ and $s^{''}$, we adopt the same idea recursively.

An easy way to calculate the transformation between basis functions is to use the transformation matrix, which can perform the transformation of all the basis functions one element has simultaneously. The transformation matrix for the right child is denoted as C_r, left as C_l. For example, Fig 1.4(b) and (c) shows the transformation matrix C_r from the right child s_r to s for linear and quadratic polynomial degree, respectively. The illustrated transformation is based on the standard triangle.

The content above has already been implemented in AMDiS. The special assembling process is called “Virtual Mesh Assembling” [51]. From a technical point of view, in

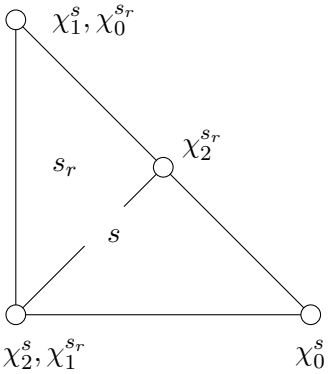
order to save unnecessary memory usage, the pair of elements from different meshes is provided in a virtual way. Furthermore, there is no need of redundant calculation of C_l and C_r at runtime. Since they cost nearly no space, C_l and C_r can be pre-calculated and stored in global static variables for all the combinations of polynomial degrees and dimensions. The multi-mesh approach is well defined and implemented on top of the AMDiS codes and it can be extended to other adaptive finite element codes with minimal effort. For more information on this topic, we refer to [51] and in Section 2.1, a more general multi-mesh approach will be introduced.

(a)



$$\chi_0^s = \chi_0^{s_l} + \frac{1}{2}\chi_1^{s_l}$$

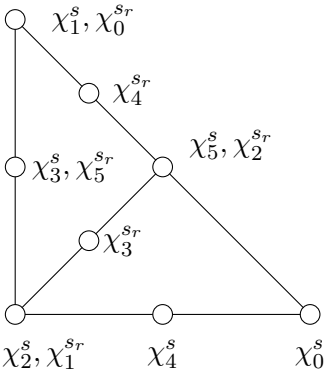
(b)



$$C_r \begin{pmatrix} \chi_0^{s_r} = -x + y \\ \chi_1^{s_r} = 1 - x - y \\ \chi_2^{s_r} = 2x \end{pmatrix} = \begin{pmatrix} \chi_0^s = x \\ \chi_1^s = y \\ \chi_2^s = 1 - x - y \end{pmatrix}$$

$$C_r = \begin{pmatrix} 0 & 0 & 0.5 \\ 1 & 0 & 0.5 \\ 0 & 1 & 0 \end{pmatrix}$$

(c)



$$C_r \begin{pmatrix} \chi_0^{s_r} = x - y + 2x^2 - 4xy + 2y^2 \\ \chi_1^{s_r} = 1 - 3x - 3y + 2x^2 + 4xy + 2y^2 \\ \chi_2^{s_r} = -2x + 8x^2 \\ \chi_3^{s_r} = 8x - 8x^2 - 8xy \\ \chi_4^{s_r} = -8x^2 + 8xy \\ \chi_5^{s_r} = -4x + 4y + 4x^2 - 4y^2 \end{pmatrix} =$$

$$\begin{pmatrix} \chi_0^s = -x + 2x^2 \\ \chi_1^s = -y + 2y^2 \\ \chi_2^s = 1 - 3x - 3y + 2x^2 + 4xy + 2y^2 \\ \chi_3^s = 4y - 4xy - 4y^2 \\ \chi_4^s = 4x - 4x^2 - 4xy \\ \chi_5^s = 4xy \end{pmatrix}$$

$$C_r = \begin{pmatrix} 0 & 0 & 0 & -0.125 & -0.125 & 0 \\ 1 & 0 & 0 & -0.125 & 0.375 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 1 \\ 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0.25 & 0.75 & 0 \end{pmatrix}$$

Figure 1.4: (a) In 1D, the basis function χ_0^s restricted to the left child s_l is represented by the linear combination of the local basis functions of the left child χ^{s_l}. (b) In 2D, the basis functions of triangle s are represented by the linear combination of the local basis functions χ^{s_r} of the right child s_r. The corresponding transformation matrix C_r is shown on the right. (c) the same situation as (b) but with quadratic polynomial degree.

CHAPTER 2

General multi-mesh method with arbitrary number of meshes

In Section 1.3, we have introduced variants of the multi-mesh method. Their implementation varies from one to another, but the basic ideas are almost the same, and they are all restricted to two-component applications. However, more than two components are common in multi-physics problems, e.g. multi-component reaction diffusion problems and multi-phase flow problems, etc. Each component, in principle, is allowed to be defined on an individual mesh. Furthermore, in some mathematics models where multiple problems are coupled together, it is practical to assign each problem an independently refined mesh, while within the same problem, a single mesh is shared by all the components just like the standard method. Here, again, the number of coupled problems does not have to be two. The situation becomes more interesting if coefficient function spaces are also involved in the system besides the test and trial function spaces. In the multi-mesh case, it is possible that one of the coefficient function spaces lives on a different mesh from the test and trial function spaces, and it is not clear how to evaluate integrals under this circumstance. In none of the existing publications, the above issues are addressed. Our aim is to make the multi-mesh method a general approach for multi-physics problems, thus all the loopholes should be plugged. In this chapter, we present a general multi-mesh method in Section 2.1, including several sub-topics, e.g. the multi-traversal algorithm (Section 2.1.2) and the solution for coefficient function spaces (Section 2.1.4). Section 2.2, on the other hand, gives more information on the software concepts and the implementation issues.

2.1 General multi-mesh concept

2.1.1 Coupling terms in the system of PDEs

To illustrate the multi-mesh technique, we consider a reaction-diffusion problem, which is used to simulate the change of the concentration of chemical substances in space Ω and time t. The problem contains three substances, denoted as $u = u(x,t)$, $v = v(x,t)$ and $w = w(x,t)$. Substances u and v react with each other and substance w reacts with both u and v. Besides that, function f is part of the reaction of u, and function g is part of the reaction of v. The equations are shown as follows:

$$\partial_t u = \nabla \cdot (\nabla u) + vu + f(u) \qquad \text{in } \Omega \times [0,\infty) \tag{2.1a}$$
$$\partial_t v = \nabla \cdot (\nabla v) + uv + g(v) \qquad \text{in } \Omega \times [0,\infty) \tag{2.1b}$$
$$\partial_t w = \nabla \cdot (\nabla w) + wuv \qquad \text{in } \Omega \times [0,\infty) \tag{2.1c}$$

with $u = 0$, $v = 0$, $w = 0$ on $\partial\Omega \times [0,\infty)$ and the initial conditions $u(t = 0) = u_0$, $v(t = 0) = v_0$, $w(t = 0) = w_0$ in $\overline{\Omega}$. We use the same way of discretization as described in Section 1.1.2. Additionally, we use the semi-implicit Euler method to perform time discretization. For the simplicity, we use the old solutions from the last iteration to approximate the true values of f and g. The above equations change to: for $n = 0, 1, 2, ..., \infty$:

$$\frac{u_{n+1}}{\tau} - \nabla \cdot (\nabla u_{n+1}) - v_{n+1}u_{n+1} = f(u_n) + \frac{u_n}{\tau} \qquad \text{in } \Omega \times [0,\infty) \tag{2.2a}$$
$$\frac{v_{n+1}}{\tau} - \nabla \cdot (\nabla v_{n+1}) - u_{n+1}v_{n+1} = g(v_n) + \frac{v_n}{\tau} \qquad \text{in } \Omega \times [0,\infty) \tag{2.2b}$$
$$\frac{w_{n+1}}{\tau} - \nabla \cdot (\nabla w_{n+1}) - w_{n+1}u_{n+1}v_{n+1} = \frac{w_n}{\tau} \qquad \text{in } \Omega \times [0,\infty) \tag{2.2c}$$

Then, we choose to linearize the term $v_{n+1}u_{n+1}$ in Eq. (2.2a) as $u_n v_{n+1}$, the term $u_{n+1}v_{n+1}$ in Eq. (2.2b) as $v_n u_{n+1}$, and the term $w_{n+1}u_{n+1}v_{n+1}$ in Eq. (2.2c) as $w_n v_n u_{n+1}$. The standard variational formulation of this system is: for $n = 0, 1, 2, ..., \infty$,

find $(u_{n+1}, v_{n+1}, w_{n+1}) \in H_0^1(\Omega) \times H_0^1(\Omega) \times H_0^1(\Omega)$ such that:

$$\int_\Omega \frac{u_{n+1}}{\tau}\phi \; dx + \int_\Omega \nabla u_{n+1} \cdot \nabla \phi \; dx - \int_\Omega u_n v_{n+1} \phi \; dx = \int_\Omega f(u_n)\phi \; dx + \int_\Omega \frac{u_n}{\tau}\phi \; dx \tag{2.3a}$$

$$\int_\Omega \frac{v_{n+1}}{\tau}\psi \; dx + \int_\Omega \nabla v_{n+1} \cdot \nabla \psi \; dx - \int_\Omega v_n u_{n+1} \psi \; dx = \int_\Omega g(v_n)\psi \; dx + \int_\Omega \frac{v_n}{\tau}\psi \; dx \tag{2.3b}$$

$$\int_\Omega \frac{w_{n+1}}{\tau}\vartheta \; dx + \int_\Omega \nabla w_{n+1} \cdot \nabla \vartheta \; dx - \int_\Omega w_n v_n u_{n+1} \vartheta \; dx = \int_\Omega \frac{w_n}{\tau}\vartheta \; dx \tag{2.3c}$$

$$\forall \phi \in H_0^1(\Omega), \forall \psi \in H_0^1(\Omega), \forall \vartheta \in H_0^1(\Omega)$$

Moreover, we adopt the multi-mesh method during space discretization. Three different meshes, S_0, S_1 and S_2, are derived from the same domain Ω and are used for u, v and w respectively. We define $\mathcal{V}^0$, $\mathcal{V}^1$ and $\mathcal{V}^2$ to be the spaces of piecewise polynomials defined on S_0, S_1 and S_2: $\mathcal{V}^i = \{v_h : v_h \in H_0^1, v_h|_s \in P^n \; \forall s \in S_i\}$ with $i = 0, 1, 2$. The question changes to: for $n = 0, 1, 2, ..., \infty$, find $(u_{h_{n+1}}, v_{h_{n+1}}, w_{h_{n+1}}) \in \mathcal{V}^0 \times \mathcal{V}^1 \times \mathcal{V}^2$ such that:

$$\int_\Omega \frac{u_{h_{n+1}}}{\tau}\phi \; dx + \int_\Omega \nabla u_{h_{n+1}} \nabla \phi \; dx - \int_\Omega u_{h_n} v_{h_{n+1}} \phi \; dx = \int_\Omega f(u_{h_n})\phi \; dx + \int_\Omega \frac{u_{h_n}}{\tau}\phi \; dx \tag{2.4a}$$

$$\int_\Omega \frac{v_{h_{n+1}}}{\tau}\psi \; dx + \int_\Omega \nabla v_{h_{n+1}} \nabla \psi \; dx - \int_\Omega v_{h_n} u_{h_{n+1}} \psi \; dx = \int_\Omega g(v_{h_n})\psi \; dx + \int_\Omega \frac{v_{h_n}}{\tau}\psi \; dx \tag{2.4b}$$

$$\int_\Omega \frac{w_{h_{n+1}}}{\tau}\vartheta \; dx + \int_\Omega \nabla w_{h_{n+1}} \nabla \vartheta \; dx - \int_\Omega w_{h_n} v_{h_n} u_{h_{n+1}} \vartheta \; dx = \int_\Omega \frac{w_{h_n}}{\tau}\vartheta \; dx \tag{2.4c}$$

$$\forall \phi \in \mathcal{V}^0, \forall \psi \in \mathcal{V}^1, \forall \vartheta \in \mathcal{V}^2$$

Then, we define the basis of $\mathcal{V}^0$, $\mathcal{V}^1$ and $\mathcal{V}^2$ as $\{\chi_1^{S_0}, ..., \chi_{m_0}^{S_0}\}$, $\{\chi_1^{S_1}, ..., \chi_{m_1}^{S_1}\}$ and $\{\chi_1^{S_2}, ..., \chi_{m_2}^{S_2}\}$. The approximate solutions can be written as the linear combinations: $u_h(x) = \sum_{i=1}^{m_0} c_i \chi_i^{S_0}(x)$, $v_h(x) = \sum_{i=1}^{m_1} d_i \chi_i^{S_1}(x)$ and $w_h(x) = \sum_{i=1}^{m_2} e_i \chi_i^{S_2}(x)$. The question changes to: for $n =$

$0, 1, 2, ..., \infty$, find $(u_{h_{n+1}}, v_{h_{n+1}}, w_{h_{n+1}}) \in \mathcal{V}^0 \times \mathcal{V}^1 \times \mathcal{V}^2$ such that:

$$\int_\Omega \frac{u_{h_{n+1}}}{\tau}\chi_i^{S_0}\ dx + \int_\Omega \nabla u_{h_{n+1}} \nabla \chi_i^{S_0}\ dx - \int_\Omega u_{h_n} v_{h_{n+1}} \chi_i^{S_0}\ dx = \int_\Omega f(u_{h_n})\chi_i^{S_0}\ dx + \int_\Omega \frac{u_{h_n}}{\tau}\chi_i^{S_0}\ dx \qquad i = 1, ..., m_0 \tag{2.5a}$$

$$\int_\Omega \frac{v_{h_{n+1}}}{\tau}\chi_i^{S_1}\ dx + \int_\Omega \nabla v_{h_{n+1}} \nabla \chi_i^{S_1}\ dx - \int_\Omega v_{h_n} u_{h_{n+1}} \chi_i^{S_1}\ dx = \int_\Omega g(v_{h_n})\chi_i^{S_1}\ dx + \int_\Omega \frac{v_{h_n}}{\tau}\chi_i^{S_1}\ dx \qquad i = 1, ..., m_1 \tag{2.5b}$$

$$\int_\Omega \frac{w_{h_{n+1}}}{\tau}\chi_i^{S_2}\ dx + \int_\Omega \nabla w_{h_{n+1}} \nabla \chi_i^{S_2}\ dx - \int_\Omega w_{h_n} v_{h_n} u_{h_{n+1}} \chi_i^{S_2}\ dx = \int_\Omega \frac{w_{h_n}}{\tau}\chi_i^{S_2}\ dx \qquad i = 1, ..., m_2 \tag{2.5c}$$

Finally, we replace the approximate solutions by the basis functions. The question again changes to: for each time iteration, find the unknown coefficients c_i, d_i and e_i such that:

$$\sum_{j=1}^{m_0} c_j \left(\sum_{s \in S_0} \int_s \frac{\chi_j^{S_0}}{\tau}\chi_i^{S_0} \right) + \sum_{j=1}^{m_0} c_j \left(\sum_{s \in S_0} \int_s \nabla \chi_j^{S_0} \cdot \nabla \chi_i^{S_0} \right) - \sum_{j=1}^{m_1} d_j \left(\sum \int_{\{s_0, s_1\}} u_{h_n} \chi_j^{S_1} \chi_i^{S_0} \right) = \sum_{s \in S_0} \int_s f(u_{h_n})\chi_i^{S_0} + \sum_{s \in S_0} \int_s \frac{u_{h_n}}{\tau}\chi_i^{S_0} \qquad i = 1, ..., m_0 \tag{2.6a}$$

$$\sum_{j=1}^{m_1} d_j \left(\sum_{s \in S_1} \int_s \frac{\chi_j^{S_1}}{\tau}\chi_i^{S_1} \right) + \sum_{j=1}^{m_1} d_j \left(\sum_{s \in S_1} \int_s \nabla \chi_j^{S_1} \cdot \nabla \chi_i^{S_1} \right) - \sum_{j=1}^{m_0} c_j \left(\sum \int_{\{s_0, s_1\}} v_{h_n} \chi_j^{S_0} \chi_i^{S_1} \right) = \sum_{s \in S_1} \int_s g(v_{h_n})\chi_i^{S_1} + \sum_{s \in S_1} \int_s \frac{v_{h_n}}{\tau}\chi_i^{S_1} \qquad i = 1, ..., m_1 \tag{2.6b}$$

$$\sum_{j=1}^{m_2} e_j \left(\sum_{s \in S_2} \int_s \frac{\chi_j^{S_2}}{\tau}\chi_i^{S_2} \right) + \sum_{j=1}^{m_2} e_j \left(\sum_{s \in S_2} \int_s \nabla \chi_j^{S_2} \cdot \nabla \chi_i^{S_2} \right) - \sum_{j=1}^{m_0} c_j \left(\sum \int_{\{s_0, s_1, s_2\}} w_{h_n} v_{h_n} \chi_j^{S_0} \chi_i^{S_2} \right) = \sum_{s \in S_2} \int_s \frac{w_{h_n}}{\tau}\chi_i^{S_2} \qquad i = 1, ..., m_2 \tag{2.6c}$$

We can see from the above equations that all the integrals resulting from the diffusion terms live on the same mesh, either S_0, S_1 or S_2, while the integrals resulting from the coupling reaction terms are defined on the union of elements from different meshes. The reaction term in Eq. (2.1a) produces an integral over S_0 and S_1. The reaction term in Eq. (2.1b) produces an integral over S_1 and S_2. Note that the most complex situation comes from the reaction term in Eq. (2.1c), whose resulting integral crosses over three meshes. The additional mesh besides S_0 and S_2 comes from the coefficient functions $w_{h_n} v_{h_n}$, which contains v_{h_n}. And according to our definition, v_{h_n} is a coefficient function living on mesh S_1 (see more in Section 2.1.4). From the given example, we see that the evaluation of the integral is possible to be defined on any subset of the meshes, depending on the original partial differential equations, which needs to be handled in a different way than the standard method.

2.1.2 Multi-mesh traversal

We denote the integral over m meshes as $\sum \int_{\{s_0,s_1,...,s_{m-1}\}}$. Based on the fact that the integral over an element s can be replaced by the sum of the integrals over its sub-parts s' and s'', if $s = s' \cup s''$, we can evaluate $\sum \int_{\{s_0,s_1,...,s_{m-1}\}}$ on the finest mesh: $\sum \int_{\{s_0,s_1,...,s_{m-1}\}} = \sum \int_{s_*}$, where $s_* = min\{s_0, s_1, ..., s_{m-1}\}$ in terms of the size of elements. This is only possible under the assumption that all meshes come from the same macro mesh and they are refined by standard refinement algorithms. The assumption is exactly the same as the two-mesh case described in Section 1.3.2. Then, to build the relationship between the elements from different meshes, we introduced a so-called multi-mesh traversal algorithm. Fig 2.1 shows a three-mesh traversal example, which is implemented based on the combination of three synchronized pre-order traversals. Note that we use both color and font to distinguish the elements from different meshes. First, the algorithm goes from the root element to the left sub-tree until a leaf element is found. The resulting element union in the first iteration is $\{s_0, s_1, s_2\} = \{0, 0, \underline{0}\}$. Then, we know that the right child of the element on S_0 corresponding to element $\underline{0}$ has not been traversed. So as the element on S_1. Thus, meshes S_0 and S_1 will move to their next elements, while $\underline{S_2}$ stays at element $\underline{0}$. The resulting element union in the second iteration is $\{s_0, s_1, s_2\} = \{1, 1, \underline{0}\}$ and the algorithm will continue until all the leaf elements are traversed. The multi-mesh traversal is a generalization of the dual-traversal introduced by T.Witkowski [51], but we introduced new data structures and algorithms to make the traversal as efficient as possible, see Section 2.2.

From a global point of view, there are two ways to perform the multi-mesh traversal: either we perform the multi-mesh traversal for each operator term one after another, creating an union of elements for each operator term separately, or we traverse all the meshes at once for all terms. We adopt the second idea due to performance reasons.

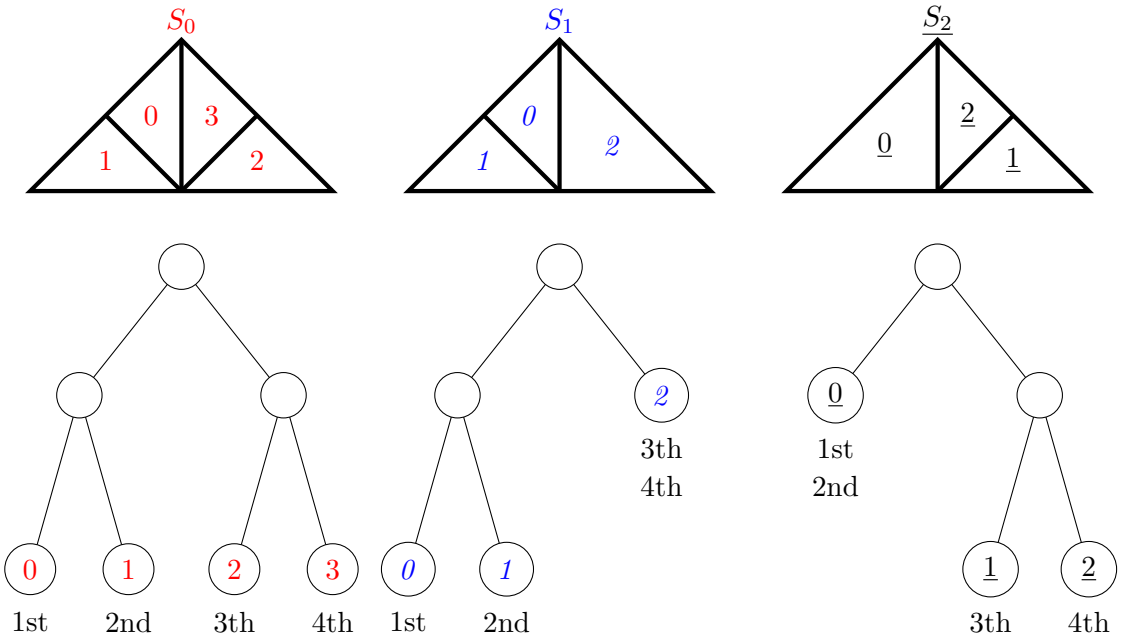


Figure 2.1: A simple example of the multi-mesh traversal with three meshes S_0, S_1 and S_2 is shown. All the meshes consist only of one macro triangle and they are refined with different refinement sets. The corresponding binary trees and the order of the traversed elements in each iteration are shown at the bottom.

2.1.3 Transformation matrices

In Section 1.3.2, we have already introduced the concept of the transformation matrix, denoted as C. During the assembling process, we need to compute the transformation matrices not only between parents and their direct children, but also between those indirect element-pairs. One efficient way to calculate the matrices recusively is to take the usage of the refinement paths between these elements. The idea is presented first by T.Witkowski [51]. Formally, an element pair can be defined by the tuple:

$$(s_1, s_2) = (s_2, \{\alpha_0, ..., \alpha_n\}) = (s_2, \alpha) \quad \alpha_i \in \{L, R\}$$

where s_2 is the larger element of the pair and α is the refinement path from s_2 to the smaller element s_1. We use L to denote a left child and R a right child. Then, the

transformation matrix is defined by:

$$C(\emptyset) = I$$
$$C(\{\alpha_0, ..., \alpha_n\}) = \begin{cases} C_L \cdot C(\{\alpha_1, ..., \alpha_n\}), & \text{if } \alpha_0 = L \\ C_R \cdot C(\{\alpha_1, ..., \alpha_n\}), & \text{if } \alpha_0 = R \end{cases} \tag{2.7}$$

The refinement path α is calculated during the multi-mesh traversal. The traversal itself has no information on the operator terms, so it has no idea which elements are needed for the evaluations of integrals. But it does not mean that we have to store the refinement paths between any two elements in the union $\{s_0, s_1, ..., s_{m-1}\}$. As already discussed, our idea is to calculate the integral on the finest mesh and to replace the basis functions on larger elements by the linear combination of the basis functions on the finest element s_*. So we only need to store the refinement paths from all the other elements to s_*, and this is calculated in each iteration of the traversal.

2.1.4 Coefficient function spaces

In this section, we focus on the situation, where coefficient function spaces live on a different mesh from either the test or trial function space. In general, the overall coefficient, denoted as ξ, in the AMDiS environment can be regard as a function (a random operator combination) of a vector of coefficient functions:

$$\xi(x) = \tilde{\xi}(\xi_0(x), \xi_1(x), ...) \tag{2.8}$$

where $\tilde{\xi}$ is the operator function. If we take the integral $\int_{\{s_0,s_1,s_2\}} w_{h_n} v_{h_n} \chi_j^{S_0} \chi_i^{S_2}$ from Eq. (2.6c) as an example, then $\xi_0(x) = w_{h_n}$, $\xi_1(x) = v_{h_n}$ and $\tilde{\xi}$ here is only the multiplication. The evaluation of $\xi(x)$ is done using the quadrature rules:

$$\int_s \xi(x) \approx \sum_k w_k \xi(x_k) \tag{2.9}$$

And if we insert Eq. (2.8) inside Eq. (2.9), we get:

$$\int_s \xi(x) \approx \sum_k w_k \tilde{\xi}(\xi_0(x_k), \xi_1(x_k), ...) \tag{2.10}$$

with:

$$\xi_i(x_k) = \sum_j \xi_i^j \chi_j^{S(i)}(\lambda^{s(i)}(x_k)), \quad S(i) \in \{S_0, S_1, ..., S_{m-1}\} \tag{2.11}$$

Each coefficient function $\xi_i(x_k)$ is a vector of coefficient related to some function bases, denoted as $\chi_j^{S(i)}$, which is defined on the barycentric coordinate of element $s(i)$, denoted

as $\lambda^{s(i)}$. And $s(i)$ can live on any mesh. We assume that we have m meshes here. But the problem is that we do not have $\lambda^{s(i)}$. Since we always evaluate on the finest element s_*, we only have the barycentric coordinate of the finest element λ^{s_*}. There are two possibilities to solve the problem. The first one is that we transform $\lambda^{s(i)}$ to the global coordinate and then transform the global coordinate back to λ^{s_*}. This is possible since s_* is the smallest element, which is contained inside $s(i)$, according to our assumption. The second possibility is that we replace $\chi_j^{S(i)}$ by the linear combination of the basis functions on s_*:

$$\chi_j^{S(i)}(\lambda^{s(i)}) = \sum_k d_k \chi_k^{s_*}(\lambda^{s_*}) \tag{2.12}$$

, where d_k is the coefficient. And this can be done using the transformation matrices directly. Since we already stored the transformation matrices for all the elements in the element union $\{s_0, ..., s_{m-1}\}$ during the multi-mesh traversal algorithm, we can apply them without additional calculation. That's the reason why we adopt the second approach. Using this strategy, we ensure that the coefficient function spaces $\xi(x)$ can be handled efficiently and we concentrate only on the test and trial function spaces in the next section.

2.1.5 Test and trial function spaces

After the element matrix, denoted as M_{el}, is calculated, we need to apply the transformation matrix C. The situation is a little bit different from the coefficient function spaces since we now need to distinguish between four cases, depending on whether or not the elements of the trial and test function spaces are the finest one s_*. Here, we take a zero order integral as an example to illustrate the deduction. The integral is denoted as $\int_{\{s_0,s_1\}} \chi_{\psi_j}^{s_1} \chi_{\phi_i}^{s_0}$, where $\chi_\phi^{s_0}$ represents the basis of the test function and $\chi_\psi^{s_1}$ the basis of the trial function.

- **CASE 1**: Both $\chi_\phi^{s_0}$ and $\chi_\psi^{s_1}$ live on the finest element s_*.
  The situation goes back to the single-mesh case, so we do not need to put any additional effort.

- **CASE 2**: Only the trial function $\chi_\psi^{s_1}$ lives on the finest element while $\chi_\phi^{s_0}$ not.
  We have $\chi_{\phi_j}^{s_0} = \sum_i^{n_B} c_{ji} \chi_{\psi_i}^{s_1}$ for $j = 0, ..., n_B$, then the result matrix M is equal to the element matrix M_{el} in the single-mesh case with an additional multiplication of the transformation matrix C from the left side. The mathematical deduction is listed below:

$$
\begin{aligned}
M &= \begin{pmatrix} \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_0}\chi^{s_0}_{\phi_0} & \dots & \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_{n_B}}\chi^{s_0}_{\phi_0} \\ \dots & \dots & \dots \\ \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_0}\chi^{s_0}_{\phi_{n_B}} & \dots & \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_{n_B}}\chi^{s_0}_{\phi_{n_B}} \end{pmatrix} \\
&= \begin{pmatrix} \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_0}\sum\limits_i (c_{0i}\chi^{s_1}_{\psi_i}) & \dots & \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_n}\sum\limits_i (c_{0i}\chi^{s_1}_{\psi_i}) \\ \dots & \dots & \dots \\ \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_0}\sum\limits_i (c_{n_Bi}\chi^{s_1}_{\psi_i}) & \dots & \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_n}\sum\limits_i (c_{n_Bi}\chi^{s_1}_{\psi_i}) \end{pmatrix} \\
&= \begin{pmatrix} c_{00} & \dots & c_{0n_B} \\ \dots & \dots & \dots \\ c_{n_B0} & \dots & c_{n_Bn_B} \end{pmatrix} * \begin{pmatrix} \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_0}\chi^{s_1}_{\psi_0} & \dots & \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_0}\chi^{s_1}_{\psi_{n_B}} \\ \dots & \dots & \dots \\ \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_{n_B}}\chi^{s_1}_{\psi_0} & \dots & \int\limits_{s_1\in S_1} \chi^{s_1}_{\psi_{n_B}}\chi^{s_1}_{\psi_{n_B}} \end{pmatrix} \\
&= C * M_{el}
\end{aligned}
$$

- **CASE 3**: Only the test function $\chi^{s_0}_{\phi}$ lives on the finest element while $\chi^{s_1}_{\psi}$ not.
  We have $\chi^{s_1}_{\psi_j} = \sum\limits_i^{n_B} c_{ji}\chi^{s_0}_{\phi_i}$ for $j = 0, ..., n_B$, then the result matrix M is equal to the element matrix M_{el} in the single-mesh method with an additional multiplication of the transpose of the transformation matrix, denoted as C^T, from the right side:

$$
\begin{aligned}
M &= \begin{pmatrix} \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_0}\chi^{s_0}_{\phi_0} & \dots & \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_{n_B}}\chi^{s_0}_{\phi_0} \\ \dots & \dots & \dots \\ \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_0}\chi^{s_0}_{\phi_{n_B}} & \dots & \int\limits_{\{s_0,s_1\}} \chi^{s_1}_{\psi_{n_B}}\chi^{s_0}_{\phi_{n_B}} \end{pmatrix} \\
&= \begin{pmatrix} \int\limits_{s_0\in S_0} \sum\limits_i (c_{0i}\chi^{s_0}_{\phi_i})\chi^{s_0}_{\phi_0} & \dots & \int\limits_{s_0\in S_0} \sum\limits_i (c_{n_Bi}\chi^{s_0}_{\phi_i})\chi^{s_0}_{\phi_0} \\ \dots & \dots & \dots \\ \int\limits_{s_0\in S_0} \sum\limits_i (c_{0i}\chi^{s_0}_{\phi_i})\chi^{s_0}_{\phi_{n_B}} & \dots & \int\limits_{s_0\in S_0} \sum\limits_i (c_{n_Bi}\chi^{s_0}_{\phi_i})\chi^{s_0}_{\phi_{n_B}} \end{pmatrix} \\
&= \begin{pmatrix} \int\limits_{s_0\in S_0} \chi^{s_0}_{\phi_0}\chi^{s_0}_{\phi_0} & \dots & \int\limits_{s_0\in S_0} \chi^{s_0}_{\phi_0}\chi^{s_0}_{\phi_{n_B}} \\ \dots & \dots & \dots \\ \int\limits_{s_0\in S_0} \chi^{s_0}_{\phi_{n_B}}\chi^{s_0}_{\phi_0} & \dots & \int\limits_{s_0\in S_0} \chi^{s_0}_{\phi_{n_B}}\chi^{s_0}_{\phi_{n_B}} \end{pmatrix} * \begin{pmatrix} c_{00} & \dots & c_{n_B0} \\ \dots & \dots & \dots \\ c_{0n_B} & \dots & c_{n_Bn_B} \end{pmatrix} \\
&= M_{el} * C^T
\end{aligned}
$$

- **CASE 4**: Neither $\chi^{s_0}_{\phi}$ nor $\chi^{s_1}_{\psi}$ lives on the finest element.
  We denote the local basis of the finest element as $\chi^{s_*}_{\theta}$, then we have $\chi^{s_0}_{\phi_j} = \sum\limits_i c_{ji}\chi^{s_*}_{\theta_i}$ and $\chi^{s_1}_{\psi_j} = \sum\limits_i c'_{ji}\chi^{s_*}_{\theta_i}$ for $j = 0, ..., n_B$. Case 2 and 3 are actually contained in this cases:

$$
\begin{aligned}
M &= \begin{pmatrix} \int\limits_{\{s_0,s_1\}} \chi_{\psi_0}^{s_1}\chi_{\phi_0}^{s_0} & \cdots & \int\limits_{\{s_0,s_1\}} \chi_{\psi_{n_B}}^{s_1}\chi_{\phi_0}^{s_0} \\ \cdots & \cdots & \cdots \\ \int\limits_{\{s_0,s_1\}} \chi_{\psi_0}^{s_1}\chi_{\phi_{n_B}}^{s_0} & \cdots & \int\limits_{\{s_0,s_1\}} \chi_{\psi_{n_B}}^{s_1}\chi_{\phi_{n_B}}^{s_0} \end{pmatrix} \\
&= \begin{pmatrix} \int\limits_{s_*} \sum\limits_i (c'_{0i}\chi_{\theta_i}^{s_*}) \sum\limits_i (c_{0i}\chi_{\theta_i}^{s_*}) & \cdots & \int\limits_{s_*} \sum\limits_i (c'_{n_Bi}\chi_{\theta_i}^{s_*}) \sum\limits_i (c_{0i}\chi_{\theta_i}^{s_*}) \\ \cdots & \cdots & \cdots \\ \int\limits_{s_*} \sum\limits_i (c'_{0i}\chi_{\theta_i}^{s_*}) \sum\limits_i (c_{n_Bi}\chi_{\theta_i}^{s_*}) & \cdots & \int\limits_{s_*} \sum\limits_i (c'_{n_Bi}\chi_{\theta_i}^{s_*}) \sum\limits_i (c_{n_Bi}\chi_{\theta_i}^{s_*}) \end{pmatrix} \\
&= \begin{pmatrix} c_{00} & \cdots & c_{0n_B} \\ \cdots & \cdots & \cdots \\ c_{n_B0} & \cdots & c_{n_Bn_B} \end{pmatrix} * \begin{pmatrix} \int\limits_{s_*} \chi_{\theta_0}^{s_*}\chi_{\theta_0}^{s_*} & \cdots & \int\limits_{s_*} \chi_{\theta_0}^{s_*}\chi_{\theta_{n_B}}^{s_*} \\ \cdots & \cdots & \cdots \\ \int\limits_{s_*} \chi_{\theta_{n_B}}^{s_*}\chi_{\theta_0}^{s_*} & \cdots & \int\limits_{s_*} \chi_{\theta_{n_B}}^{s_*}\chi_{\theta_{n_B}}^{s_*} \end{pmatrix} * \begin{pmatrix} c'_{00} & \cdots & c'_{n_B0} \\ \cdots & \cdots & \cdots \\ c'_{0n_B} & \cdots & c'_{n_Bn_B} \end{pmatrix} \\
&= C * M_{el} * C'^T
\end{aligned}
$$

For general first and second order terms where the derivative is involved, we have:

$$\nabla\chi_\phi^{s_0} = \nabla(\sum_i w_i\chi_{\theta_i}^{s_*}) = \sum_i \nabla(w_i\chi_{\theta_i}^{s_*}) = \sum_i w_i\nabla\chi_{\theta_i}^{s_*}$$

Because these operations are linear operations, the rules of zero order terms are also applicable for first and second order terms, which is a very nice characteristic for us. Last but not least, we also need to take the right side of the equations into account. The situation is even simpler since only the test function space is involved. So the deduction is omitted.

In summary, we illustrate the new assembling algorithm in Algorithm 1. First of all, we assemble the operators, whose trial function, test function and coefficient functions are defined on the same mesh, by calling the standard single-mesh functions. Note that we are still under a multi-mesh traversal, it is likely that a mesh will stay on the former element although the traversal moves to the next iteration; hence, line 6 uses an additional If-statement to present the codes from duplicated computation. Then, the algorithm checks at line 11 whether there is an operator, whose test and trial functions live on different meshes, or at least one of the coefficients lives on a different mesh. If this is the case, the presented multi-mesh assembling is used for those operators and it is shown in Algorithm 2. We did not give the single-mesh assembling algorithm, but if the basic ideas introduced in the previous sections are clear, one can easily find out that line $5-6$ and $10-17$ are the additional steps for the multi-mesh method.

Algorithm 1 General assembling in the multi-mesh case

Require: *stack*: traverse stack of all meshes, *matrix*: stiffness matrix

```
1: while stack is empty == false do
2:   for all matrix do
3:     get test function mesh and trial function mesh from matrix
4:     get coefficient function mesh from operator
5:     if test mesh == trial function mesh == coefficient mesh then
6:       if a new element on mesh then
7:         assemble non-coupled terms of matrix
8:         assemble non-coupled terms of vector
9:       end if
10:    end if
11:    if coupling term exists then
12:      assemble coupling terms of matrix
13:      assemble coupling terms of vector
14:    end if
15:  end for
16: end while
```

2.2 Software concepts

In this section, we show how the multi-mesh concept is implemented in AMDiS. We start from the multi-mesh traversal. In order to record the traversed elements of meshes in each iteration, we introduced a list of elements, with each element in the list coming from an individual mesh. The concept is similar to the union of elements $\{s_0, s_1, ..., s_{m-1}\}$ described in Section 2.1.2. The only difference is that the elements in the list are sorted by their refinement levels in descending order. As a consequence, the elements with the smallest difference in volume are neighbors in the list. We will give the explanation later. We have already known that some meshes might stay on the former elements from the last iteration. The decision, whether a mesh goes to the next element s_{next} or stays at the former one, denoted as s_{curr}, is related to the volume of elements. If s_{curr} has the smallest volume $s_{curr} = s_*$, in the next iteration, the corresponding mesh will update its status (replace the former element by a new one). If not, the corresponding mesh of s_{curr} stays until the sum of the volume of the traversed elements of the finer neighbor (the mesh where s_{small} lives on) is equal to the volume of s_{curr}. The notation of the elements is shown in Fig 2.2. If we do not sort the list, we have to search, which element is s_{small}, for each element in the list. Another benefit of the sorted list is the accessibility of the finest element s_* since it always appears at the

Algorithm 2 Assembling of the coupling terms

Require: s^i: element list of the current iteration, *matrix*: stiffness matrix

get the finest element s_{el} in s^i
get the mesh S_{el} and the element stiffness matrix M_{el} of s_{el}
for all *term* on *matrix* **do**
  for all coefficient of vectors in term **do**
    get transformation matrix C^{co}
    evaluate the integral on s_{el} using C^{co}
  end for
  calculate and sum up all the integrals to M_{el}
end for
if test mesh != S_{el} **then**
  get transformation matrix C
  $M_{el} = C * M_{el}$
end if
if trial mesh != S_{el} **then**
  get transformation matrix C'
  $M_{el} = M_{el} * C'^T$
end if
add M_{el} to *matrix*

head of the list.

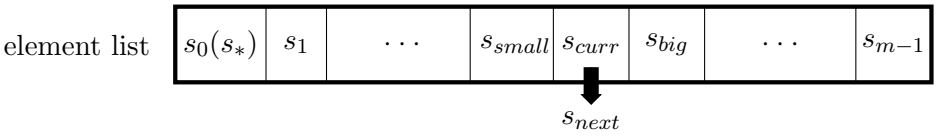


Figure 2.2: Element notations in an element list of m meshes
s_{curr}: the current element under discussion
s_{small}: the element in the list, which has the next smaller volume of s_{curr}
s_{next}: the next element to be traversed on the same mesh as s_{curr}
s_*: the finest element in the list

From the implementation point of view, the comparison of the element volume is based on the refinement level of elements. We denote $l(i)$ as the refinement level of the ith element in the list. Then, for each mesh, it holds a remaining volume denoted as

RV, which is calculated during each iteration as follows:

$$RV = \begin{cases} 0 & \text{if } i = 0 \\ RV - \frac{1}{2^{l(i-1)-l(i)}} & \text{if } i > 0 \end{cases} \tag{2.13}$$

The initial value of RV is set to 1. Status update is triggered when RV goes to zero. Fig 2.3 shows a two-mesh example of the calculation of RV. The remaining volume RV of the left-side mesh is always zero since the corresponding element is always the finer one, so the traversal on the left-side mesh moves to the next element in each iteration. We are interested in, when the traversal on the right-side mesh goes from the left element to the right one, and the answer is already shown in the figure. Note that the number shown in the element is the corresponding refinement level. When there are more than two meshes in the list, a reverse iterator is used to check RV from the end of the element list because if the coarser mesh moves to the next element, so as the meshes, which are finer or equal to the coarse one.

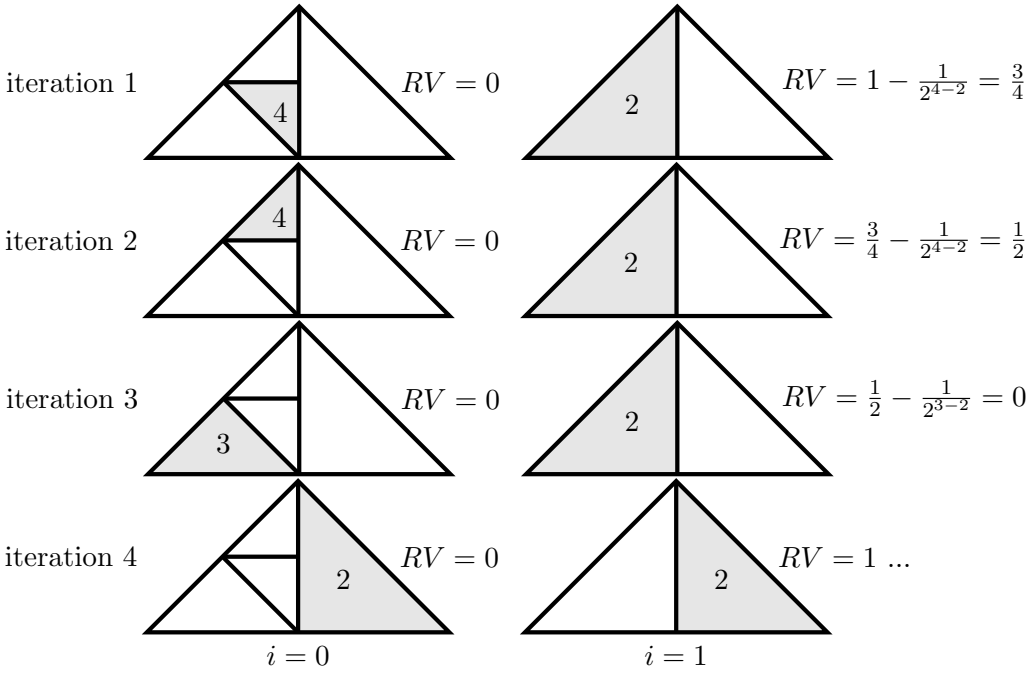


Figure 2.3: Two-mesh example of the volume comparison method based on the refinement level of elements

Now, our task is to implement the above concepts based on the existing codes. In AMDiS, the traversal of a binary tree is managed by the class `TraverseStack`. Any element data that can be computed during the mesh traversal will not be stored

in the memory. These data include the Jacobian of the barycentric coordinates, the vertex coordinates, the neighbor information, etc. Which kind of information is collected depends on the given assembling flag. During the traversal, the requested data are computed and written into an object called `ElInfo`. Our multi-mesh traversal is mainly based on `TraverseStack` and `ElInfo`. The new classes we introduced are as follows (see also the class diagram in Fig 2.4):

- `MultiTraverse` is the main class for users to manage the multi-mesh traversal. It provides almost the same interface as `TraverseStack` does. Internally, it contains a vector of `TraverseStacks` for each mesh. It is this class that handles the status of traversed meshes using a `MultiElInfo` object.

- `MultiElInfo` is responsible for the counting of RV. It stores a sorted list of `ElInfos`, but it stores also an unsorted `ElInfo` list and a mesh index list, which are only used internally. To conclude, this class serves as a helper class used for the multi-mesh traversal.

- `ElInfos` is a generalized version of `ElInfo`, which can be created from `MultiElInfo` by its get_sorted member function. `ElInfos` inherits only the sorted `ElInfo` list and it marks the mesh of the test and trial function spaces by member variables row_idx and col_idx or by functions row and col. This class serves as a lightweight interface object for other functions and classes such as the member function assemble of `DOFVector` and `DOFMatrix`, getElementMatrix of `Operator`, calculateElementMatrix of `Assembler` and so on.

A code comparison between the single- and the multi-mesh traversals is shown in Fig 2.5. The user interface is almost the same. In the multi-mesh case, we call

```
Traverse stack;
ElInfo *elInfo = stack.traverseFirst(mesh, level, flag);
while (elInfo) {
  // do work with elInfo
  elInfo = stack.traverseNext(elInfo);
}
```

```
MultiTraverse stack;
MultiElInfo multiElInfo;
stack.setFillSubElemMat(true, basisFcts);
bool cont = stack.traverseFirst(meshes, levels, flags, multiElInfo);
while (cont) {
  // do work with multiElInfo
  cont = stack.traverseNext(multiElInfo);
}
```

Figure 2.5: Single and multi Traversal usage example

traverseFirst from `MultiTraverse` with the parameters: a vector of meshes, the levels of the meshes to be traversed and the assembling flags. `MultiTraverse` provides an interface named setFillSubElemMat. In most instances, the first parameter is set to true, indicating that we want to calculate the transformation matrices between larger elements and the smallest one during the traversal. This depends on the basis functions that are used, which are given in the second parameter. setFillSubElemMat should be called before the start of the traversal. Then, in the traversal loop body, each `ElInfo` can call getSubElemCoordsMat to get its transformation matrix.

Besides the multi-mesh traversal, the way of the implementation of the transformation matrix is also a crucial point in terms of computing efficiency. As already discussed, we stored the transformation matrices between direct children and parents in static variables. For those indirect element-pairs, we implemented a software cache to store transformation matrices since the calculation of transformation matrices can considerably increase the time of the assembling process if there is an extremely large gap between the refinement levels. The cache itself is stored in `ElInfo` regardless of the dimension. This approach is a trade-off between time and space, but the overall memory usage of the cache is around 2 MB in all our tests, thus there is no need to implement an upper bound and a replacement method for the cache. Algorithm 3 of class `ElInfo` shows how to obtain a transformation matrix from the refinement path. From the implementation point of view, a refinement path is actually a 64 bit integer data type, which stores 1 for a right child and 0 for a left child, bitwise. Note that in the algorithm, the recursive computation of the transformation matrix starts from the

end of the refinement path. That is why C_r and C_l are multiplied from the right side, which looks opposite but actually the same as described in Eq. (2.7).

Algorithm 3 `ElInfo`.getSubElemCoordsMat

Require: *basisFct*: a function pointer of the basis function, *transformMatrices*: the cache, *path*: the refinement path

```
1:  if path not ∈ transformMatrices then
2:    C = I
3:    C_l = transformMatrices[0]
4:    C_r = transformMatrices[1]
5:    for i = 0; i < path.length; i++ do
6:      if path & (1 << i) then
7:        C = C * C_r
8:      else
9:        C = C * C_l
10:     end if
11:   end for
12:   transformMatrices[path] = C
13: end if
14: return transformMatrices[path]
```

2.3 Summary

One restriction of our multi-mesh method is that the linear combination replacement of the basis functions is limited to the same polynomial degree. This is clear since it is impossible to replace a quadratic basis function by the linear combination of linear basis functions. So the multi-mesh method is not suitable for the Navier-Stokes equations with a standard Taylor-Hood element, i.e. second order Lagrange finite elements for the velocity and linear Lagrange finite elements for the pressure. One alternative is that we use standard linear finite elements for both fields, but with a different mesh. In 2D, the mesh for the velocity is refined twice more than the mesh for the pressure. In 3D, the velocity mesh has to be refined three times to get the corresponding refinement structure [6].

The general multi-mesh method provides the possibility for handling more than two meshes, but at the cost of a more complicated assembling process. The calculation and the application of the transformation matrix and the multi-mesh traversal spend additional time. So, whether or not this method will bring us performance speedup depends on the degree the number of DOFs can be saved. In principle, the larger the difference between the refinement sets of the meshes involved in the system, the better

the performance simulations can achieve. Sometimes, users need to use the multi-mesh traversal outside the assembling. Then, the number of times the multi-mesh traversal is called should be as small as possible. In the best case, everything can be performed within one single traversal. This is a good habit since the traversal might take quite a long time when the number of meshes is relatively large. To conclude, there is no doubt that the multi-mesh approach will bring benefits if applications are appropriate and users are careful with their implementation.

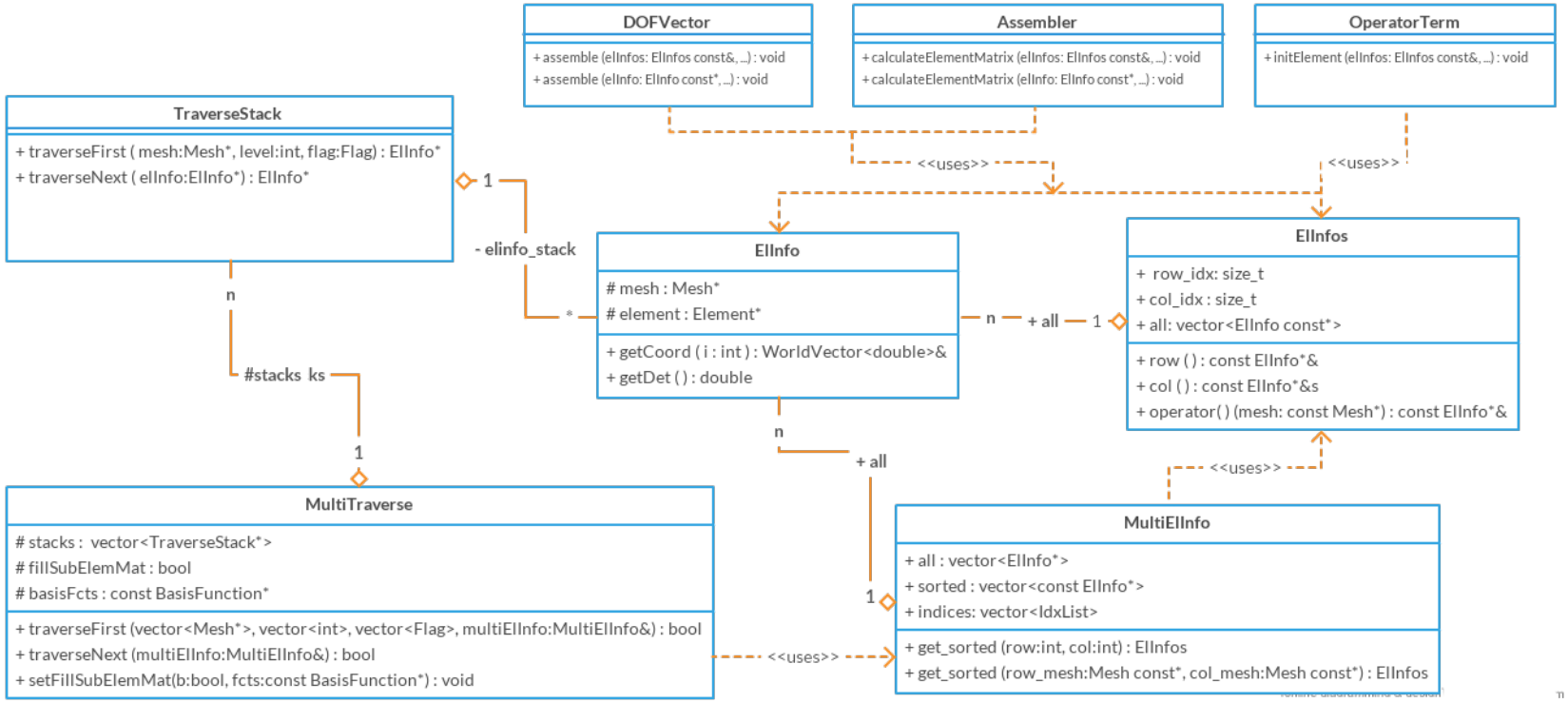


Figure 2.4: Class diagram of the classes related to multi-mesh traversal

CHAPTER 3

Parallel multi-mesh Concept

Parallelization of the finite element codes became quite popular in the eighties to make best use of available computing platforms: clusters, grids, etc. These infrastructures can be distinguished into two main categories by the organization of physical memory [37]. The first category is made up of physically shared memory systems, where all processors have direct access to the same memory and data communication is done by shared variables. The second one is made up of physically distributed memory systems, where all processors have a private memory and data communication between each other is done via some message protocols, e.g. message passing interface (MPI) [41]. But there exist also many hybrid organizations, for example, a virtually shared memory on top of a physically distributed memory system. The very first numerical approaches are based on the shared memory systems. But the memory size of a single processor seems to be too small to be sufficient with the growth of problem sizes. So in recent years, the tendency is to increase the number of cores on the chip, where each core has a private memory space.

Nowadays, high performance computing (HPC) systems contain computing nodes consisting of multiple CPUs, which by themselves include multiple cores. Computing nodes such as graphics processing units (GPUs) that consist of several hundreds or even thousands of relatively simple cores are quite common. Typically, these systems are based on a programming model called "single program, multiple data" (SPMD), i.e. multiple processors simultaneously execute the same program with different data. The simultaneous execution does not mean that each line of codes is executed synchronized between processors, though. Additional parallel communications and synchronizations are necessary.

The domain decomposition method is one of the most popular SPMD models used for the finite element method. It fits well into the distributed memory architecture

[46, 11]. The entire computational domain is split into several sub-domains, each of which is distributed to one core. Most of the work on sub-domains is independent from other sub-domains, thus computation can be performed in parallel to obtain performance speedup. Domain decomposition methods are classified to two groups: overlapping ones, e.g. the Schwarz alternating method and the additive Schwarz method [12], and non-overlapping ones, e.g. finite element tearing and interconnect (FETI), Dual-Primal FETI Method (FETI-DP) [17, 26, 13] and the balancing domain decomposition (BDDC) [30, 31, 32]. In AMDiS, we adopt the non-overlapping domain decomposition strategy [52, 50].

To our best knowledge, we are the first to introduce the multi-mesh method into the parallel environment. In Section 3.1, we present our main idea of the parallel multi-mesh from the perspective of the adaption loop. Section 3.2 introduces the parallel DOF mapping, which is used for the DOF enumeration. In Section 3.3, the software concepts and the algorithms are introduced in detail. Though, our method is implemented in C++, none of the presented data structures or algorithms is limited to a specific programming language. The general concepts are portable to other FEM toolboxes.

3.1 Parallel multi-mesh adaption loop

The domain decomposition method leads to not only the challenge of data partition and distribution, but also the challenge of the redefinition of numerical algorithms. For the first challenge, we use a mesh consisting of a subset of the macro elements to represent a sub-domain, and use local DOF matrices and vectors to represent the local stiffness matrices and vectors. For the second challenge, we need to reconsider those algorithms inside the adaption loop. Most of them are relatively simple to be parallelized. Assembler, error estimator and marker can first be performed on each local processor independently, and then be followed by an MPI gather communication over all sub-domains. But the situation is totally different for solvers. Parallel solvers require a globally successive enumeration of local data in order to establish the global linear system. In the multi-mesh case, the construction of the globally successive enumeration becomes even more complicated since now data come from different meshes. The parallel management of multiple meshes also makes the algorithms more error-prone. In this section, we first introduce the main idea of our parallel multi-mesh method, beginning with a parallel multi-mesh adaption loop. For comparison, the parallel single-mesh adaption loop is shown in Table 3.1.

The parallel adaption loop contains additional parallel steps compared to the former version introduced in Section 1.1.3. At the beginning of the simulation, the domain decomposition is performed on each processor by reading the whole domain from the macro file and then cutting off the macro elements, which do not belong to themselves.

1: initialize parallelization
2: assemble, solve, estimate
3: **while** tolerance not reached **do**
4: adapt mesh
5: **if** load out of balance **then**
6: repartition mesh
7: **end if**
8: assemble, solve, estimate
9: **end while**

Table 3.1: Single-mesh parallel adaption loop

Which macro elements belong to which processors is hinted by third party partitioners. In AMDiS, we use Zoltan [8, 9] and ParMETIS [25, 24, 38].

If the initial tolerance is not reached, local adaptive refinement is performed on sub-domains. Local processors have no information on the refinement of their neighbors, so it is likely that the refinement sets along the left and the right sides of the sub-domain boundary do not match with each other, generating hanging nodes from a global point of view. In order to prevent this from happening, a check of the boundaries between every two processors is used in parallel, directly after local refinement. If the check successfully detects the hanging nodes, one side of the sub-domain boundary, which has a coarser refinement set, will be refined so that it has the same refinement hierarchy as the opposite side. This procedure is called "parallel mesh adaption", or simply "parallel adaption", which is implemented via the mesh structure code introduced in Section 1.2.2.1. Because of the parallel adaption, a parallel mesh in AMDiS is never coarser than its sequential version.

If the distribution of workload gets out of balance after the completion of the parallel adaption, users have the opportunity to use the mesh repartitioning algorithm to assign a more balanced partition for their simulations. But the mesh repartitioning itself causes much heavier communications compared to the parallel adaption, because not only the boundary but also the entire refinement structures of the macro elements need to be transferred between processors. So it is not practical to perform the repartitioning as soon as possible. An alternative way is to perform the repartitioning if the load keeps imbalanced for a certain number of iterations, e.g. 25 or even more.

Based on the above knowledge, we present a parallel multi-mesh adaption loop. First of all, we recall the prerequisite of the multi-mesh method mentioned in 2.1.1: the multi-mesh method is only feasible if the meshes are derived from the same macro mesh. So similarly, if we want to run the multi-mesh method on multiple processors, each

processor should have identical subsets of the macro mesh, i.e. meshes must share the same partition, from the very beginning. Otherwise, we have to migrate macro elements between processors before the assembling process, which only wastes parallel resources. Then, the local mesh adaption is performed on each mesh, and each processor checks the interior boundaries one mesh after another. The imbalancing factor is now calculated via the sum of the number of leaf elements of all the meshes one processor has. If the load is not balanced, just like the parallel initialization, a new partition is shared for all meshes among all processors. The repartitioning of meshes is also preformed one after another. Last but not least, the interface of solvers needs to be modified to allow DOFs from different meshes. To conclude, we do not need any modification on the content of the multi-mesh method since it is a pure local method. The difficulty of the parallel multi-mesh method lies in, how to handle multiple meshes in parallel. Table 3.2 shows the parallel multi-mesh adaption loop, where changes are marked underlined.

1: initialize parallelization
2: **multi assemble**, solve, estimate
3: **while** tolerance not reached **do**
4: **independently** adapt meshes
5: **if** load out of balance **then**
6: repartition meshes
7: **end if**
8: **multi assemble**, solve, estimate
9: **end while**

Table 3.2: Multi-mesh parallel adaption loop

3.2 Parallel DOF enumeration

As mentioned before, we use the non-overlapping domain decomposition method in parallel. The resulting sub-domains intersect only on the interface. The DOFs located on the interface is shared between sub-domains. Within each sub-domain, interface DOFs exist in its local linear system. But from the perspective of the parallel solver, the local interface DOFs, that correspond to the same global DOF, along with their contributions, are merged together in the global linear system. In our convention, we set the owner of the interface DOFs to be the sub-domain, which has a higher processor ID.

In AMDiS, we have introduced three different sets of indices for the DOFs. The finite element method requires the first one to assemble local matrices and vectors, which is called local index set. The indices in this set must be enumerated with a continuous

sequence. For those solver methods, which cannot be applied in a pure local way, we need to enumerate all the DOFs across sub-domains to get the second index set, named global index set. The global index set must satisfy the condition that the local indices of those interface DOFs, corresponding to the identical global DOFs, are mapped into the same global indices. The global index set is continuous across sub-domains. Third, when there is more than one component involved in the system of PDEs, we need to distinguish those DOFs that belong to different components. The new index set, which takes multiple components into consideration, is called matrix index set. The matrix index is the final global linear system index, which is unique over all sub-domains and components. In AMDiS, "Parallel DOF mapping" is responsible for the mapping from local DOF indices to matrix ones, or from global DOF indices to matrix ones.

If the finite element spaces have the same polynomial degree, we only need to store one copy of the DOF mapping to build up the matrix index set. When mixed finite elements are used for different components, different DOF mappings have to be used for each component separately. The parallel management of multiple meshes is also responsible for the enumeration of the DOFs that belong to different meshes. The main difference between the single- and the multi-mesh methods in the process of the DOF enumeration is that even if two components are discretized by finite element spaces with the same polynomial degree, they cannot share a common DOF mapping if they are defined on different meshes. This is clear since DOFs on different meshes are by definition different DOFs and the components might have completely different sets of DOFs if the corresponding mesh refinement hierarchies are different.

The enumeration rules are as follows: for each sub-domain, we define $D_i = \{1, ..., d_i\}$ to be the set of all DOF indices in processor i. The subset $\bar{D}_i \subset D_i$ contains all DOF indices that are owned by the current processor. We denote the number of DOFs in $\bar{D}_i$ as $nRank_i = |\bar{D}_i|$. We assume that $\bar{D}_i$ is also a continuous set of indices to simplify the following definitions. Furthermore, we denote the smallest global index of DOFs on rank i as $rStart_i$, which is defined by:

$$rStart_i = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{p=0}^{i-1} nRank_p & \text{if } i > 0 \end{cases} \tag{3.1}$$

In terms of the global indices, in order to establish the relationship between the interface DOFs which correspond to the same global DOF, we define the mapping $\mathcal{R}_i^j(d) = e, d \in D_i, e \in D_j$ if local indices d in D_i and e in D_j have such relationship. Then, the global indices are defined as:

$$globalIndex(i, d) = \begin{cases} rStart_i + d & \text{if } d \in \bar{D}_i \\ globalIndex(j, d') & \text{if } d \notin \bar{D}_i, \mathcal{R}_i^j(d) = d' \end{cases} \tag{3.2}$$

In terms of the matrix indices, we take multiple components into account. We denote $rRank_i$ on component j to be $rRank_i^j$, $rStart_i$ on component j to be $rStart_i^j$. There are two possibilities to traverse the DOFs: either we first traverse the DOFs of component 1 on all processors, and then go to component $2, 3, ...$, or we first traverse all the DOFs processor 1 has for all the components, and then go to processor $2, 3, ...$ and so on. In AMDiS, we chose the second possibility. Then, the matrix index of component j on processor i can be written as:

$$matIndex(i,j,d) = \begin{cases} \sum_{p=1}^{n} rStart_i^p + \sum_{p=1}^{j-1} nRank_i^p + d & \text{if } d \in \bar{D}_i \\ matIndex(k,j,d') & \text{if } d \notin \bar{D}_i, \mathcal{R}_i^k(d) = d' \end{cases} \tag{3.3}$$

where n is the total number of components. A simple DOF indices enumeration example is shown in Fig 3.1, where two components with linear finite element spaces are defined on meshes S_0 and S_1 separately. Note that the three DOF index sets are distinguished by different colors.

3.3 Software concepts

In parallel, distributed meshes are the input data, and the linear system is the output data. Building a path between the input and output data is the main job of the parallel codes. We already introduced the strategy of the parallel DOF enumeration, but it is not the whole story. In this section, we want to go deeper into the implementation.

3.3.1 Parallel data containers

We start with the introduction of the basic data structures [52] used in parallel. These classes are mainly data containers, which carry information about domain partition, boundaries and communication data. The most important classes are introduced below:

- `MeshDistributor` is in charge of moving the macro mesh elements between processors using mesh structure codes. Besides that, nearly all the parallel work is encapsulated inside the class, e.g. parallel initialization, mesh repartitioning, etc.

- `MeshPartitioner` is responsible for the management of a partition map, whose keys indicate macro element indices and value processor IDs. As long as we have the mapping, we can move macro elements, together with their refinement sets and associated values, to the specified processors via `MeshDistributor`. In order to create the map, AMDiS internally transforms the AMDiS mesh to the mesh structures recognizable by third-party partition libraries.

- `ElementObjectDatabase` uses the partition map from `MeshPartitioner` to record the ownership information for each macro element. Then, it breaks down the ownership information to the level of vertices, edges or faces, which are used by `InteriorBoundary`.

- `InteriorBoundary` creates a set of boundary objects for each sub-domain. A boundary object is a geometry part of an element shared across processors. The boundary objects are subdivided into two groups: those that are owned by the processor and those that are only part of the sub-domain but owned by other processors. After `InteriorBoundary` is created, we can build the data to be communicated between processors. `InteriorBoundary` is stored on each processor although the mesh is partitioned.

- `DofComm` object can be used to synchronize the DOF values between neighbor subdomains via point-to-point communication. It has the knowledge of the interface DOFs, but it has no global view on the entire domain. DOFs from different finite element spaces can be handled within one `DofComm` as long as the finite spaces live on the same mesh. Assuming that an `InteriorBoundary` is already initialized, we can easily create a `DofComm` object without any additional communication.

- `ParallelDofMapping` is the implementation of the strategy of the parallel DOF enumeration discussed in Section 3.2. In most cases, each `ParallelSolver` contains one `ParallelDofMapping`. Typically, if a parallel solver asks for the DOF indices mapping, it will register its `ParallelDofMapping` to `MeshDistributor` in its constructor.

- `ParallelSolver` is the end point of the information flow, which is a general solver interface provided for a large class of parallel solvers, for example, "Portable, Extensible Toolkit for Scientific Computation" (PETSc) [5, 4, 3] and "Matrix Template Library" (MTL) [19, 18]. Child classes of this class can transform the linear system from the AMDiS format to their own formats and solve the linear system inside the inherited function solveLinearSystem.

For more review, I refer to T.Witkowski [52]. Among all the classes, `MeshDistributor` is the most important. On one hand, it is the only class users get in touch with, if they want to run their simulations in parallel. On the other hand, except for `ParallelDofMapping`, all the other above software concepts have association or aggregation relationship with `MeshDistributor`. To get a better understanding, we subdivide the data containers inside `MeshDistributor` into two groups: those only related to macro meshes and those related to DOFs. Fig 3.2 shows the information flow through parallel data structures. The classes above the dashed line, `MeshPartitioner`, `ElementObjectDatabase` and `InteriorBoundary`, belong

to the first group, which need a rebuilding after the mesh repartitioning, while the classes below the dashed line, `DofComm`, `ParallelDofMapping` and `BoundaryDofInfo`, are DOF-related structures, which need a rebuilding after the mesh adaption.

For the first group, the situation is relatively simple since the carried information is identical for all meshes, e.g. the macro boundaries between sub-domains. Most of the algorithms in the first group can be directly reused. We only need to store for each mesh the pointers to itself and to the macro elements. The second group is responsible for the management of DOFs. `BoundaryDofInfo` records the geometrical information about boundary DOFs. We will not discuss it in detail since it is only used by some specific solvers. `DofComm` can handle all the DOFs defined on the same mesh regardless of the finite element spaces. Now, we have multiple meshes, an easy modification is to use a map between the meshes and the `DofComm` objects. However, this method is not suitable for `ParallelDofMapping` since we want to have exactly one matrix index set at the end, so we created a nested class, named `ComponentDofMap`, inside `ParallelDofMapping` to strictly separate the DOF mappings between components, that are defined on different meshes. Later, solvers can access the DOF mapping easily via `ComponentDofMap` to build the global linear system. To sum up, Fig 3.3 lists the class diagram of the classes discussed above.

3.3.2 Parallel algorithms

- checkMeshChange (Algorithm 4)

Algorithm 4 `MeshDistributor`.checkMeshChange

Require: *meshes*: vector of mesh pointer
1: **for all** *mesh* in *meshes* **do**
2: **if** *mesh* changes from last iteration **then**
3: **repeat**
4: parallel mesh adaption
5: **until** no more refinement on *mesh*
6: updateDofRelatedStruct(*mesh*)
7: **end if**
8: **end for**
9: MPI_Barrier()
10: updateLocalGlobalNumbering()
11: **if** load is not balanced **then**
12: repartition all the meshes
13: **end if**

Now, we move our concentration to the redefinition of parallel algorithms. We have already shown some of the important member functions of `MeshDistributor` in the class diagram Fig 3.3. In this section, we use the function checkMeshChange with all its sub-functions, as a concrete example to illustrate how the management of multiple meshes is implemented in AMDiS.

One of the main tasks of checkMeshChange is the parallel mesh adaption. But it does more than just fixing hanging nodes. When the additional parallel mesh refinement on the interior boundaries causes the emergence of new DOFs, it should be reflected in the final large sparse linear system. Till line 8, all the meshes have preformed their recursive refinement so that the meshes are guaranteed to be conforming meshes. The sub-function updateDofRelatedStruct inside the loop will be discussed below. From the beginning of line 11, we check the load-balancing and if the imbalancing factor is above the threshold, the mesh repartitioning is performed, as mentioned before.

- updateDofRelatedStruct (Algorithm 5)

Algorithm 5 `MeshDistributor`.updateDofRelatedStruct

Require: *mesh*: mesh pointer

1: *mesh*.dofCompress()
2: createBoundaryDofs(*mesh*)
3: updateDofsToDofMapping(*mesh*)

The parallel mesh adaption invalidates the DOF-related data structures. So updateDofRelatedStruct is in charge of updating the DOF containers by flushing the DOFs, which are newly created, into the containers. At line 1 we call dofCompress of the mesh to eliminate all holes of unused DOF indices for each finite element space, more details see S.Vey [48]. After dofCompress, we call createBoundaryDofs to recreate the `DofComm` and if necessary, the `BoundaryDofInfo` as well. At the end, we flush all the DOFs to `ParallelDofMapping` by updateDofsToDofMapping.

- updateDofsToDofMapping (Algorithm 6)

Algorithm 6 `MeshDistributor`.updateDofsToDofMapping

Require: *mesh*: mesh pointer, *dofMaps*: vector of parallel DOF mapping

```
1: for all dofMap in dofMaps do
2:   get all finite element spaces dofMapSpaces defined on mesh in dofMap
3:   if dofMapSpaces not empty then
4:     get dofComm from dofMap
5:     dofMap.clear(mesh)
6:     for feSpace in dofMapSpaces do
7:       get DOF_all belonging to this feSpace
8:       get DOF_recv from dofComm[feSpace]
9:       dofMap[feSpace].insertRankDof(DOF_recv)
10:      dofMap[feSpace].insertNonRankDof(DOF_all − DOF_recv)
11:    end for
12:  end if
13: end for
```

In updateDofsToDofMapping, we first traverse all the `ParallelDofMapping`s, which have been registered in `MeshDistributor`. Then, if the finite element space of the stored `ComponentDofMap` object inside `ParallelDofMapping` lives on the exact same mesh as the one given in the function parameter, we need to update this `ComponentDofMap`. In line $7 - 10$, we start to traverse all the DOFs on the finite element space, denoted as *feSpace*. During traversal, we distinguish between rank DOFs and non-rank ones and flush them into separate groups for the simplicity of the usage of our DOF enumeration method. This can easily be done by a `DofComm` object. That is why updateDofsToDofMapping is called after createBoundaryDofs in checkMeshChange. The DOF set which includes all the DOFs belonging to *feSpace* is denoted as DOF_{all}. It is available via getAllDofs of the mesh, and the DOFs received from DOF communicators: DOF_{recv} are by definition non-rank DOFs. Then, rank DOFs are calculated by $DOF_{all} - DOF_{recv}$. After updateDofsToDofMapping is finished, both `DofComm` and `ParallelDofMapping` are up-to-date.

- updateLocalGlobalNumbering (Algorithm 7)

Algorithm 7 `MeshDistributor`.updateLocalGlobalNumbering

Require: *dofMaps*: vector of parallel DOF mapping

```
1: for all dofMap in dofMaps do
2:    dofMap.update()
3: end for
4: if has periodic boundary then
5:    create periodic DOF mappings
6:    for all dofMap in dofMaps do
7:       dofMap.computeMatIndex()
8:    end for
9: end if
```

The last sub-function in checkMeshChange we have not discussed yet is updateLocalGlobalNumbering, which is responsible for the DOF enumeration. Note that there is a MPI barrier operation across all sub-domains before we call updateLocalGlobalNumbering in checkMeshChange. This step cannot be omitted since the parallel DOF enumeration only makes sense after all the sub-domains have already updated their `ParallelDofMappings`. updateLocalGlobalNumbering starts the enumeration for each `ParallelDofMapping` by calling its member function update. Line $4 - 9$ are only used when a periodic boundary condition exists.

- update (Algorithm 8)

Algorithm 8 `ParallelDofMapping`.update

Require: *mesh*: mesh pointer, *compDofMaps*: vector of component DOF mappings

```
1: for all compDofMap in compDofMaps do
2:    if mesh of compDofMap == mesh then
3:       compDofMap.update()
4:    end if
5: end for
6: nRankDofs = computeRankDofs()
7: nLocalDofs = computeLocalDofs()
8: nOverallDofs = computeOverallDofs()
9: rStartDofs = computeStartDofs()
10: computeMatIndex()
```

This is a member function of `ParallelDofMapping`, which is almost the last function triggered by checkMeshChange. Inside, each `ComponentDofMap` object will

call its own update to calculate the individual $nRank$ and $rStart$ (see Section 3.2) by MPI group communications and also the global index set via sub-function computeGlobalMapping. Then, in line $6-9$, the overall $nRank$, $rStart$ are calculated by gathering the data from each component, on the basis of which, the most important matrix index set is computed via computeMatIndex. Indices of rank DOFs can be calculated directly by Eq. (3.3), while indices of non-rank DOFs are calculated by their owner processors, and then exchanged via `DofComm` to other processors. Finally, a new globally successive DOF index set over all sub-domains is established after the parallel mesh adaption.

3.4 Summary

In this chapter, we introduced the basic idea of the parallel multi-mesh method. The idea comes from the combination of the mesh adaption loop and the pre-conditions of our multi-mesh method. Then, we pointed out the difficulty in handling multiple meshes in parallel, especially the difficulty in the management of the degrees of freedom associated with the meshes. To solve this problem, we presented the parallel DOF enumeration strategy. In the last section, we showed our detailed implementation of the parallel data structures and the algorithms. Here, instead of concentrating on the distribution and the communication of meshes, we discussed more on the establishment of the global linear system, which includes one very important function: checkMeshChange. Although it is impossible to show everything within one single function, it does cover the most critical content we want to explain.

Note that the presented strategy of the load-balancing is not perfect, especially for those simulations, where problems with individual solvers are coupled together. Since each problem has its own linear system, the computation of the weights of sub-domains might not be useful. What we do is that we sum up the number of DOFs on all the meshes as the weights. But in the situation we discussed, the DOFs on the other meshes belong to other problems, which will not appear in the linear system of this problem. We have to admit that the overall imbalancing factor cannot reflect the true status of workload for each problem separately. Actually, we cannot expect all the problems to be optimal after assigning the same partition to them.

There is no doubt that the parallel mesh adaption and the mesh repartitioning in the multi-mesh case cost more overhead than the standard method. We are wondering, whether or not in the worst case the overhead can dominate the whole parallel performance. In the next chapter, we will show a numerical example related to this topic and at least in this example, we find out that the costs of multiple meshes will not be the bottleneck in parallel.

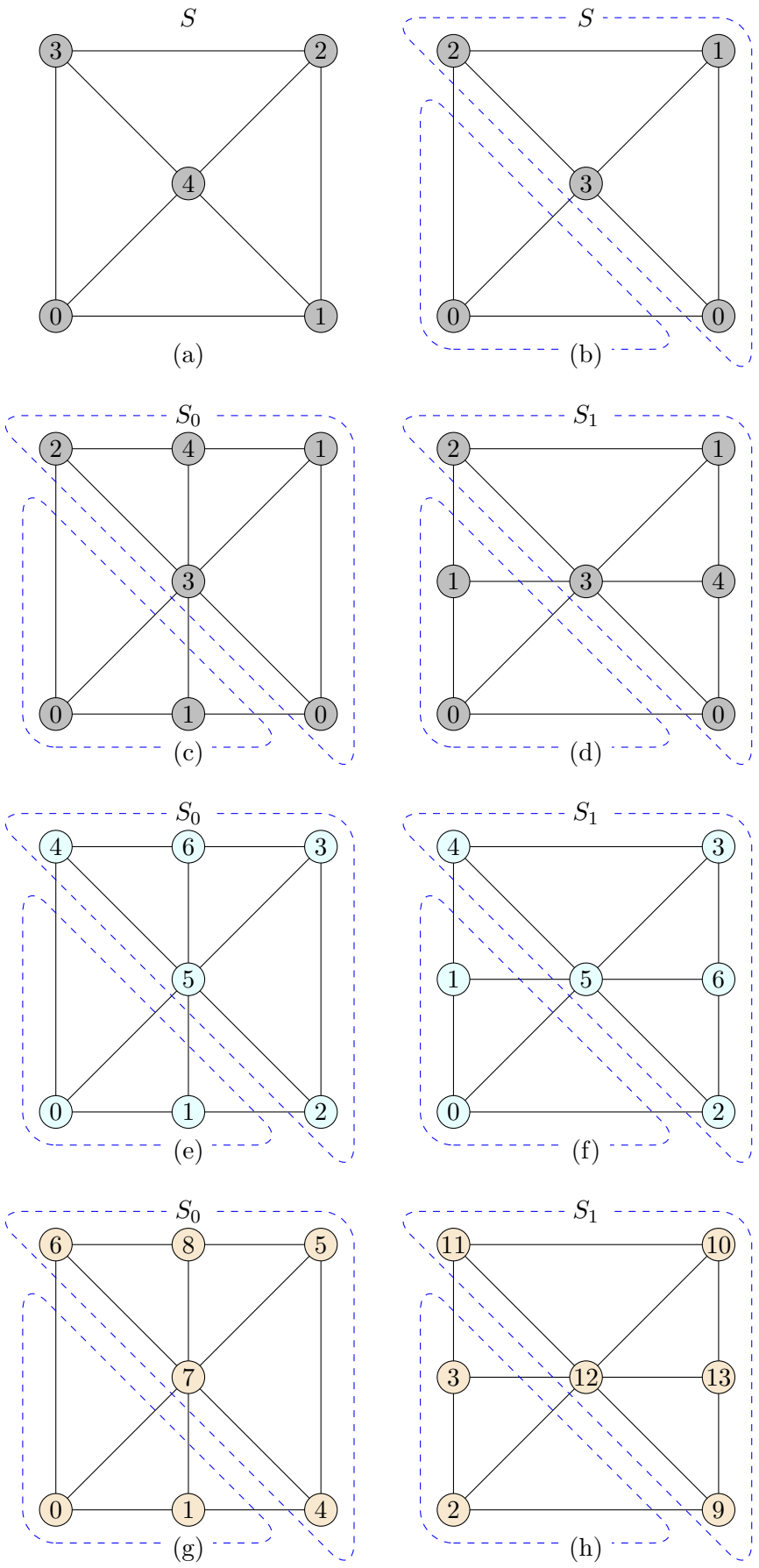

S
3
2
4
0
1
(a)
S
2
1
3
0
0
(b)
S_0
2
4
1
3
0
1
0
(c)
S_1
2
1
1
3
4
0
0
(d)
S_0
4
6
3
5
0
1
2
(e)
S_1
4
3
1
5
6
0
2
(f)
S_0
6
8
5
7
0
1
4
(g)
S_1
11
10
3
12
13
2
9
(h)

Figure 3.1: DOF indices enumeration in different stages: (a) local indices on macro mesh S (b) local indices after parallelization, with two sub-domains marked by the blue dashed circles (smaller circle represents processor 1, larger processor 2) (c) local indices of component 1 defined on S_0 after local adaption (d) local indices of component 2 defined on S_1 after local adaption (e) global indices of component 1 (f) global indices of component 2 (g) matrix indices of component 1 (h) matrix indices of component 2
gray: local idx, bubbles: global idx, champagne: matrix idx

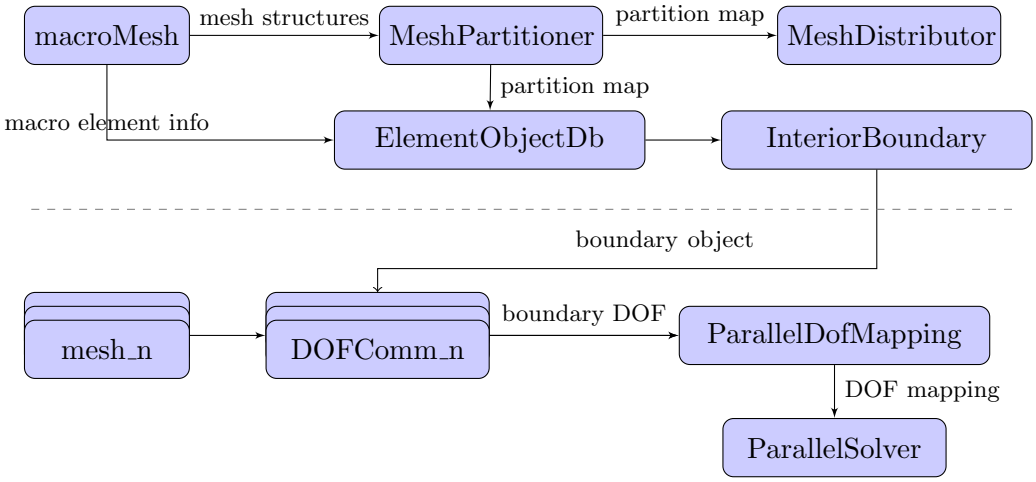


Figure 3.2: Two information flows through parallel data structures, starting from the geometry

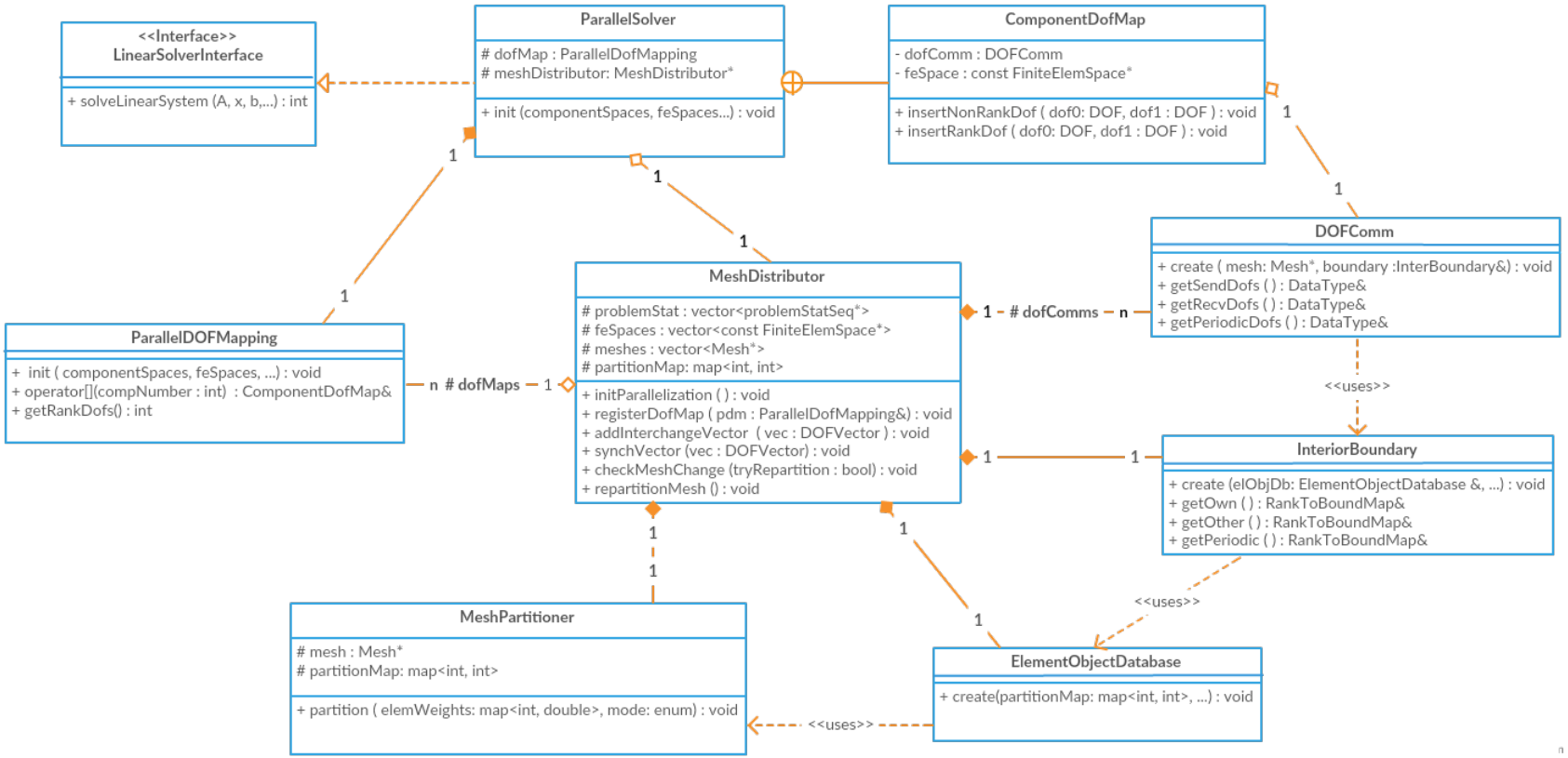


Figure 3.3: Class diagram of the main parallel classes

CHAPTER 4

Numerical experiments

To show that the multi-mesh approach is superior in contrast to the standard single-mesh finite element method, in this chapter, we illustrate some simulation examples. In Section 4.1, we consider the dynamics of deformable objects in flow. These objects could be bubbles, cells or particles. In all cases, the objects are fluid like with an interface surrounding them. The main content in this section comes from the publication [29]. We first introduce the hydrodynamic phase field model, which was originally presented by W.Marth [33]. The main idea of this model and the numerical approach can be found in Section 4.1.1 and Section 4.1.2. In terms of the mesh adaption, we found out that it is reasonable to combine this model with the multi-mesh concept presented in this thesis. The way how we achieved the combination is introduced in Section 4.1.3. Finally, in Section 4.1.4, we show that the resulting simulation can be more efficient without losing accuracy and we analyse why this is the case. By the illustration of the multi-phase flow problem, we want to emphasize the advantage of the arbitrary-number-of-mesh attribute of the multi-mesh approach.

In Section 4.2, we consider a dendritic growth problem in 2D using a phase-field model. The order of the sections is organized in the same way as the multi-phase flow problem, but here, we want to emphasize the advantage of the parallel multi-mesh method. The simulation is split into several time periods. Each period is executed by a different number of processors. We show that even in a parallel environment, the multi-mesh method is still better than the standard method.

4.1 Multi-phase flow problem

We are interested in the dynamics of deformable objects in flow. A huge class of two-phase flow problems can be formulated in this situation and has been considered by various numerical approaches, like the Navier-Stokes-Cahn-Hilliard equations and so on. Most of these studies consider only one object and its deformation is a result of the hydrodynamic interaction. The situation becomes more complex if more objects are involved. In addition to the mentioned hydrodynamic interaction, the objects also interact with each other, which requires the computation of the distance between the object. As they are deformable and thus can have complex shapes this becomes a nontrivial task.

We concentrate here on phase field models using an auxiliary variable ϕ describing the interface between the objects and the surrounding fluid implicitly. For each object i, we use a phase field variable ϕ_i and take only account for short range steric interactions. An efficient way to describe these interactions in a local manner is proposed. We will consider an example, which could be interpreted as a model for interacting red blood cells.

4.1.1 Governing equations

The diffuse interface formulation uses the auxiliary phase fields ϕ_i that distinguish the inside and the outside of each object labeled with index $i = 1, \dots, N$. The phase field variables are defined as

$$\phi_i(t, \mathbf{x}) := \tanh\left(\frac{r_i(t, \mathbf{x})}{\sqrt{2}\epsilon}\right) \tag{4.1}$$

where ϵ characterizes the thickness of the diffuse interface and $r_i(t, \mathbf{x})$ denotes the signed-distance function between $\mathbf{x} \in \Omega$, in the considered case a bounded domain in $\mathbb{R}^2$ and its nearest point on $\Gamma_i(t)$ the interface of object i. Depending on r_i we consider the inside with $\phi_i \approx 1$ and the outside with $\phi_i \approx -1$. The interface $\Gamma_i(t)$ is then implicitly defined by the zero level set of ϕ_i. We consider $\phi = \max_{\mathbf{x}\in\Omega}(\phi_1, \dots, \phi_N)$, which defines the object phase $\phi \approx 1$, the fluid phase $\phi \approx -1$, and the interfaces Γ as the zero level set of ϕ. The dynamics are governed by equations that couple these phase fields to the actual physical degrees of freedom. We consider a free energy $\mathcal{E}(\phi_1, \dots, \phi_N)$, to be specified below, and use a H^{-1}-gradient flow, which together with the convective term due to the underlying fluid velocity $\mathbf{v}$ read for each $i = 1, \dots, N$

$$\partial_t \phi_i + \mathbf{v} \cdot \nabla \phi_i = \gamma \Delta \phi_i^\natural \tag{4.2}$$

with a small positive mobility coefficient γ and the chemical potentials

$$\phi_i^\natural = \frac{\delta \mathcal{E}(\phi_1, \dots, \phi_N)}{\delta \phi_i}. \tag{4.3}$$

The extended incompressible Navier-Stokes equation, which accounts for a local inextensibility constraint ($\nabla_\Gamma \cdot \mathbf{v} = 0$ on Γ) reads

$$\rho(\partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v}) + \nabla p - \frac{1}{\mathrm{Re}} \nabla \cdot (\nu \mathbf{D}) = \sum_{i=1}^{N} \phi_i^\natural \nabla \phi_i + \nabla \cdot (\frac{|\nabla \phi|}{2} \mathbf{P} \lambda_{local}) + \mathbf{F} \tag{4.4}$$

$$\nabla \cdot \mathbf{v} = 0 \tag{4.5}$$

with density $\rho = 1$, viscosity $\nu = (1-\phi)/2 + \sum_{i=1}^{N} (\nu_i/\nu_0)(\phi_i + 1)/2$, deformation tensor $\mathbf{D} = \nabla \mathbf{v} + (\nabla \mathbf{v})^T$, pressure p, projection operator $\mathbf{P} = \mathbf{I} - (\nabla \phi \otimes \nabla \phi)/|\nabla \phi|^2$, applied force $\mathbf{F}$ and Reynolds number $\mathrm{Re} = \rho U L/\nu_0$, with characteristic velocity U and characteristic length L. Different densities could be handled in a similar way but are omitted here for simplicity. The Lagrange multiplier λ_{local} is determined by solving the equation

$$\xi \epsilon^2 \nabla \cdot (\phi^2 \nabla \lambda_{local}) + \frac{|\nabla \phi|}{2} \mathbf{P} : \nabla \mathbf{v} = 0 \tag{4.6}$$

with $\xi > 0$ a parameter independent of ϵ. This phase-field approximation for the inextensibility constraint was introduced in [1]. The considered energy is a Helfrich-type energy for each phase field ϕ_i

$$\mathcal{E}_i(\phi_i) = \frac{1}{2\mathrm{ReBe}_i} \int_\Omega \frac{1}{\epsilon} \left(\epsilon \Delta \phi_i - \frac{1}{\epsilon} (\phi_i^2 - 1)(\phi_i + H_0) \right)^2 d\Omega, \tag{4.7}$$

with bending capillary numbers $\mathrm{Be}_i = (4/3)\sqrt{2}\nu_0 U L^2 / b_{N,i}$, with the bending rigidity $b_{N,i}$, see [14]. In addition we consider penalty energies of the form

$$\mathcal{E}_{i,area}(\phi_i) = \frac{c_i}{2\mathrm{ReBe}_i} (\mathcal{A}_i^0 - \mathcal{A}(\phi_i))^2, \tag{4.8}$$

with penalty parameters c_i, to ensure global area conservation of the interfaces. The initial and desired area of object i are denoted by $\mathcal{A}_i^0$ and $\mathcal{A}(\phi_i) = \int_\Omega \frac{\epsilon}{2} |\nabla \phi_i|^2 + \frac{1}{4\epsilon} (\phi_i^2 - 1)^2 d\Omega$, respectively. The penalty term helps to control the accumulation of errors, as shown in [15, 1].

We also require an object-object interaction energy $\mathcal{E}_{int}$ and thus the coupling of all phase field variables $\phi_1, \ldots, \phi_N$. The interaction is a function of the distances between the objects, which can become tedious to compute as the objects can have complex shapes. We here consider only steric interactions to prevent coalescence or overlapping of objects and model a short range repulsion by a Gaussian potential. Using Eq. (4.1), the signed distance function r_j, measuring the signed distance from any point in Ω to the interface Γ_j can be computed within the diffuse interface region as

$$r_j = -\frac{\epsilon}{\sqrt{2}} \ln \frac{1 + \phi_j}{1 - \phi_j} \quad \forall \mathbf{x} : |\phi_j(\mathbf{x})| < 1. \tag{4.9}$$

We thus can write the interaction potential within the phase-field description as

$$\mathcal{E}_{i,int}(\phi_1,\ldots,\phi_N) = \frac{1}{\mathrm{ReIn}}\int_\Omega B(\phi_i)\sum_{\substack{j=1\\ j\neq i}}^{N} w_j d\Omega \tag{4.10}$$

with $B(\phi_i) = \frac{1}{\epsilon}(\phi_i^2-1)^2$ being nonzero only within the diffuse interface around Γ_i, the interaction function

$$w_j = \begin{cases} \exp\left(-\frac{1}{2}(\ln\frac{1+\phi_j}{1-\phi_j})^2\right), & \text{if } |\phi_j(\mathbf{x})|<1 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

and $\mathrm{In} = (4/3)\sqrt{2}\nu_0 U/\alpha$ the interaction number, with $\alpha > 0$ the strength of the repulsive interaction. The overall interaction energy thus reads

$$\mathcal{E}_{int}(\phi_1,\ldots,\phi_N) = \sum_{i=1}^{N}\mathcal{E}_{i,int}(\phi_1,\ldots,\phi_N) \tag{4.12}$$

and we obtain

$$\mathcal{E}(\phi_1,\ldots,\phi_N) = \sum_{i=1}^{N}(\mathcal{E}_i(\phi_i)+\mathcal{E}_{i,area}(\phi_i)) + \mathcal{E}_{int}(\phi_1,\ldots,\phi_N) \tag{4.13}$$

and thus

$$\frac{\delta\mathcal{E}(\phi_1,\ldots,\phi_N)}{\delta\phi_i} = \frac{\delta\mathcal{E}_i(\phi_i)}{\delta\phi_i} + \frac{\delta\mathcal{E}_{i,area}(\phi_i)}{\delta\phi_i} + \frac{\delta\mathcal{E}_{int}(\phi_1,\ldots,\phi_N)}{\delta\phi_i} \tag{4.14}$$

with

$$\begin{aligned}
\frac{\delta\mathcal{E}_i(\phi_i)}{\delta\phi_i} &= \frac{1}{\mathrm{ReBe}_i}\psi_i \\
\psi_i &= \Delta\mu_i - \frac{1}{\epsilon^2}(3\phi_i^2 + 2H_0\phi - 1)\mu_i \\
\mu_i &= \epsilon\Delta\phi_i - \frac{1}{\epsilon}(\phi_i^2-1)(\phi_i+H_0) \\
\frac{\delta\mathcal{E}_{i,area}(\phi_i)}{\delta\phi_i} &= \frac{c_i}{\mathrm{ReBe}_i}\kappa_i(\mathcal{A}_i^0 - \mathcal{A}(\phi_i)) \\
\kappa_i &= \epsilon\Delta\phi_i - \frac{1}{\epsilon}(\phi_i^2-1)\phi_i \\
\frac{\delta\mathcal{E}_{int}(\phi_1,\ldots,\phi_N)}{\delta\phi_i} &= \frac{1}{\mathrm{ReIn}}(B'(\phi_i)\sum_{\substack{j=1\\ j\neq i}}^{N} w_j + w_i'\sum_{\substack{j=1\\ j\neq i}}^{N} B(\phi_j))
\end{aligned}$$

with

$$w_i' = \begin{cases} \frac{2}{\phi_i^2-1}\ln\frac{1+\phi_i}{1-\phi_i}\exp\left(-\frac{1}{2}(\ln\frac{1+\phi_i}{1-\phi_i})^2\right), & \text{if } |\phi_i(\mathbf{x})|<1 \\ 0 & \text{otherwise.} \end{cases}$$

The overall system to be solved consists now of the N higher order convection-reaction-diffusion equations for the phase field variables ϕ_i Eq. (4.2) with (4.3) and (4.14), the extended incompressible Navier-Stokes equations, Eq. (4.4) and (4.5) for velocity $\mathbf{v}$ and pressure p and the equation for the Lagrange multiplier λ_{local}, Eq. (4.6).

4.1.2 Numerical approach

4.1.2.1 Time discretization

In order to discretize in time, we explore an operator splitting approach. In an iterative process we first solve the flow problem and substitute its solution into the phase-field equations, which are then solved separately with a parallel splitting method. We split the time interval $I = [0, T]$ into equidistant time instants $0 = t_0 < t_1 < \ldots$ and define the time steps $\tau := t_{n+1} - t_n$. We define the discrete time derivative $d_t \cdot^{n+1} := (\cdot^{n+1} - \cdot^n)/\tau$, where the upper index denotes the time step number and e.g $\mathbf{v}^n := \mathbf{v}(t_n)$ is the value of $\mathbf{v}$ at time t_n. For each system, a semi-implicit time discretization is used, which together with an appropriate linearization of the involved non-linear terms leads to a set of linear system in each time step.

4.1.2.2 Space discretization

We apply the finite element method to discretize in space, where a P^2/P^1 Taylor-Hood element is used for the flow problem, all other quantities are discretized in space using P^2 elements. In each time step we solve:

1. the flow problem for $\mathbf{v}^{n+1}$, p^{n+1} and λ_{local}^{n+1}

$$
\begin{aligned}
& d_t \mathbf{v}^{n+1} + (\mathbf{v}^n \cdot \nabla)\mathbf{v}^{n+1} = \\
& \quad - \nabla p^{n+1} + \frac{1}{\mathrm{Re}} \nabla \cdot (\nu^n \mathbf{D}^{n+1}) + \sum_{i=1}^{N} \phi_i^{\natural^n} \nabla \phi_i^n + \nabla \cdot (\frac{|\nabla \phi^n|}{2} \mathbf{P}^n \lambda_{local}^{n+1}) + \mathbf{F}, \\
& \nabla \cdot \mathbf{v}^{n+1} = 0 \\
& \xi \epsilon^2 \nabla \cdot ((\phi^n)^2 \nabla \lambda_{local}^{n+1}) + \frac{|\nabla \phi^n|}{2} \mathbf{P}^n : \nabla \mathbf{v}^{n+1} = 0
\end{aligned}
$$

where $\nu^n = \nu(\phi^n)$ and $\mathbf{P}^n = \mathbf{I} - \frac{\nabla \phi^n \otimes \nabla \phi^n}{|\nabla \phi^n|^2}$

2. the phase field equations for ϕ_i^{n+1}, $i = 1, \ldots, N$

$$
\begin{aligned}
d_t\phi_i^{n+1} + \mathbf{v}^{n+1} \cdot \nabla\phi_i^{n+1} &= \gamma\Delta\phi_i^{\natural\, n+1}, \\
\phi_i^{\natural\, n+1} &= \frac{1}{\mathrm{ReBe}_i}\psi_i^{n+1} + \frac{c_i}{\mathrm{ReBe}_i}(\mathcal{A}_i^0 - \mathcal{A}(\phi_i^n))\kappa_i^n \\
&\quad + \frac{1}{\mathrm{ReIn}}\left(B'(\phi_i^n)\sum_{\substack{j=1 \\ j\neq i}}^{N} w_j^n + w_i^{n'}\sum_{\substack{j=1 \\ j\neq i}}^{N} B(\phi_j^n) \right), \\
\psi_i^{n+1} &= \Delta\mu_i^{n+1} - \frac{1}{\epsilon^2}(3(\phi_i^{n+1})^2 + 2H_0\phi_i^{n+1} - 1)\mu_i^{n+1}, \\
\mu_i^{n+1} &= \epsilon\Delta\phi_i^{n+1} - \frac{1}{\epsilon}((\phi_i^{n+1})^2 - 1)(\phi_i^{n+1} + H_0)
\end{aligned}
$$

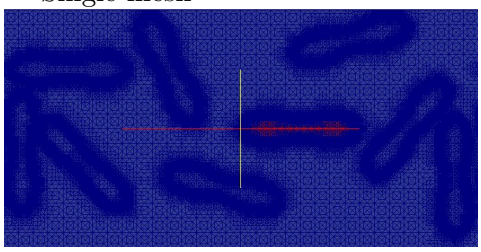
with $\kappa_i^n = \epsilon\Delta\phi_i^n - \frac{1}{\epsilon}((\phi_i^n)^2 - 1)\phi_i^n$. We linearize the non-linear terms by a Taylor expansion of order one, e.g. $((\phi_i^{n+1})^2 - 1)\phi_i^{n+1} = ((\phi_i^n)^2 - 1)\phi_i^n + (3(\phi_i^n)^2 - 1)(\phi_i^{n+1} - \phi_i^n)$.

4.1.3 Adaptivity

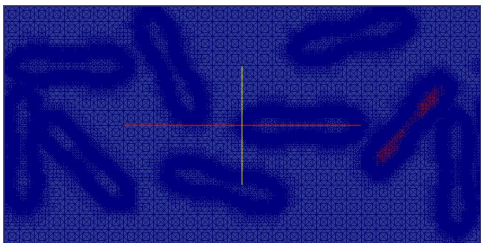
The problem involves various length scales: large scales related to the computational domain, intermediate scales related to the size of the objects, and small length scales related to the diffuse interface width, which is of order ϵ. In order to effectively resolve these length scales, we need to consider adaptive mesh refinement. We here only account for a heuristic criteria to dynamically refine the diffuse interface of each object. The criteria is a critical value for the gradient of the phase field variables $\phi_i^{\natural}$, which is adjusted to guarantee 8 grid points across the diffuse interface. An analog strategy is used for coarsening, with the coarsest possible mesh still fine enough to resolve the flow field. Other criteria, based on error estimates can be used in the same manner.

While adaptive mesh refinement has become a standard tool in phase field modeling and is one of the reasons why the approach can compete with other numerical methods to solve interface problems, there are further possibilities to improve the efficiency. When the standard single-mesh finite element method is used, we define each phase field variable $\phi_i^{n+1}, i = 0, , , N-1$ and the velocity variables v_x and v_y on the same mesh, as shown in the upper part of Fig 4.1. We can see that each phase field uses the same mesh with the refinement set to be the union of the refinement on all diffuse interfaces, among which only the interface of itself (marked in red) makes sense. The extra refinement does nothing but enlarge the final linear system. It is more reasonable to define each object on an independently refined mesh to reduce the overall computational time. So we use N meshes and refine the meshes to resolve the diffuse interface region for each phase field variable respectively. However, the refinement set of the velocity field remains the same as in the single-mesh case, as shown in the lower part of Fig 4.1.

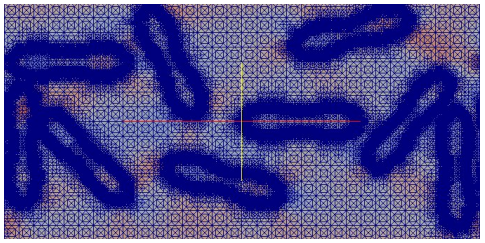
- Single mesh



ϕ_0

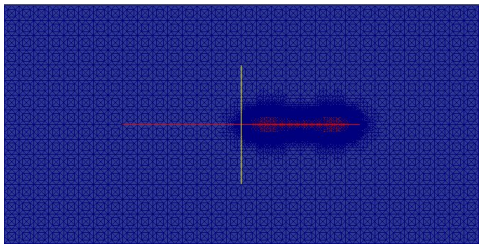


ϕ_1 ...

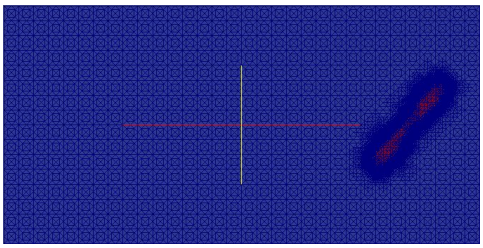


v_x (v_y)

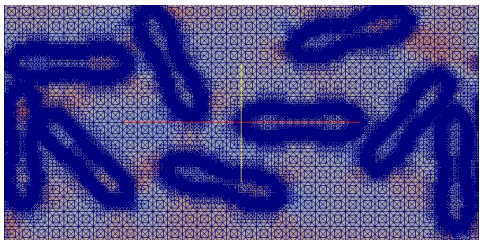
- Multi mesh



ϕ_0



ϕ_1 ...



v_x (v_y)

Figure 4.1: Mesh comparsion between single-mesh and multi-mesh approach for the phase field $\phi_i, i = 0, , , N-1$ and the fluid field v_x, v_y

As our systems involve terms with more than one variable, again we need to consider coupling terms between variables. Critical for all adaptive strategies, which involve coarsening, is the handling of the time derivative, as information will be lost if ϕ_i^n

is simply interpolated onto the new mesh on which to solve for ϕ_i^{n+1}. This can be circumvented by using the multi-mesh strategy, too.

We use the same adaptively refined mesh for the variables ϕ_i^{n+1}, $\phi_i^{\natural^{n+1}}$, ψ_i^{n+1} and μ_i^{n+1}. We denote it by S_i^{n+1}. The only critical terms, which need further treatment in the phase field equations are thus the terms

$$\mathbf{v}^{n+1} \cdot \nabla \phi_i^{n+1}, \quad B'(\phi_i^n) \sum_{\substack{j=1 \\ j \neq i}}^{N} w_j^n \quad \text{and} \quad w_i^{n'} \sum_{\substack{j=1 \\ j \neq i}}^{N} B(\phi_j^n),$$

as here $\mathbf{v}^{n+1}$, ω_j^n and ϕ_j^n are defined on differently refined meshes. For the first term, the velocity will be computed on the finest common mesh of all phase field variables, denoted by S^{n+1}. It thus coincides with S_i^{n+1} within the diffuse interface region of ϕ_i^{n+1}, but away from the interface it is significantly finer. Interpolating $\mathbf{v}^{n+1}$ onto the mesh of ϕ_i^{n+1} thus is no option, as information would be lost. We thus consider the multi-mesh strategy for this term.

For the other two terms, we can also use the strategy of multi-mesh and it provides us better performance than the standard method. But here we even have a better option than the multi-mesh approach. We demonstrate a version, where the coupling terms of variables can be handled by interpolating one variable onto the mesh of the other. Due to the special structure of the coupling terms between the phase field variables this can be handled without loss of information. What we do is that we interpolate ϕ_j^n onto the mesh of ϕ_i^{n+1}. This is correct since $B'(\phi_i^n) \approx 0$ and $\omega_i^{n'} \approx 0$ outside of the diffuse interface region of ϕ_i^n. The summation terms will thus be considered as

$$I_{S_i^n}^{S_i^{n+1}}(B'(\phi_i^n)) \sum_{\substack{j=1 \\ j \neq i}}^{N} I_{S_j^n}^{S_i^{n+1}}(w_j^n) \quad \text{and} \quad I_{S_i^n}^{S_i^{n+1}}(w_i^{n'}) \sum_{\substack{j=1 \\ j \neq i}}^{N} I_{S_j^n}^{S_i^{n+1}}(B(\phi_j^n)),$$

with the interpolation operator $I_{S_i^n}^{S_i^{n+1}} : S_i^n \to S_i^{n+1}$. The interpolation across multiple meshes is implemented based on the multi-mesh traversal algorithm.

Also in the Navier-Stokes equations we use the same adaptively refined mesh for the variables $\mathbf{v}^{n+1}$, p^{n+1} and λ_{local}^{n+1}. Also ϕ^n we will be defined on that mesh and we consider $I_{S^n}^{S^{n+1}}(\phi^n)$. The only critical term thus is

$$\sum_{i=1}^{N} \phi_i^{\natural^n} \nabla \phi_i^n \tag{4.15}$$

with all ϕ_i^n, more precisely $I_{S_i^n}^{S_i^{n+1}}(\phi_i^n)$ defined on the coarser meshes S_i^{n+1}. We thus can interpolate them on the finer Navier-Stokes mesh S^{n+1} and perform the summation on

that mesh. We thus consider

$$\sum_{i=1}^{N} I_{S_i^n}^{S_i^{n+1}}(\phi_i^{n}) I_{S_i^n}^{S_i^{n+1}}(\nabla\phi_i^n).$$

Here, the interpolation can improve the performance much more than the full multi-mesh approach, which will be explained in the next section. The only remaining terms, where the multi-mesh assembly has to be used, are the time derivative $\partial_t\phi_i^{n+1}$ and the convective term $\mathbf{v}^{n+1}\cdot\nabla\phi_i^{n+1}$. To ensure conservation of ϕ_i the terms (ϕ_i^n, ξ_i^{n+1}), with ϕ_i^n defined on S_i^n and test function ξ_i^{n+1} defined on S_i^{n+1}, has to be computed, virtually on the finest common mesh, see Section 2.1.3. This also holds for the term $(\mathbf{v}^{n+1}\cdot\nabla\phi_i^{n+1}, \xi_i^{n+1})$, which is virtually assembled on the finer Navier-Stokes mesh S^{n+1}.

4.1.4 Performance and accuracy

We consider a test case of a rectangular domain with thickness 20 μm and length 40 μm, with periodic boundary conditions on the in- and outflow, left and right, and no-slip condition on the upper and lower wall. We set Re$= 1.125\cdot 10^{-4}$, Be$_i = 5.3$, mobility $\gamma = 10^{-5}$ and a viscosity ratio $\nu_i/\nu_0 = 1$. Flow is considered through the force term $\mathbf{F} = (\frac{1}{\mathrm{Fr}}, 0)^\top$, with Froude number Fr$= 2.4\cdot 10^{-6}$ leading to a maximal velocity of magnitude 10. We randomly place N objects in the domain. They have initially an ellipsoidal shape and are arranged, such that they do not overlap. As the numerical approach for one object has been validated in detail in [1], we here only compare the solution obtained with the single-mesh and the multi-mesh approach. Fig 4.2 shows a snapshot of the 0-level line of the phase field variables at the final time in comparison with the single mesh simulation results.

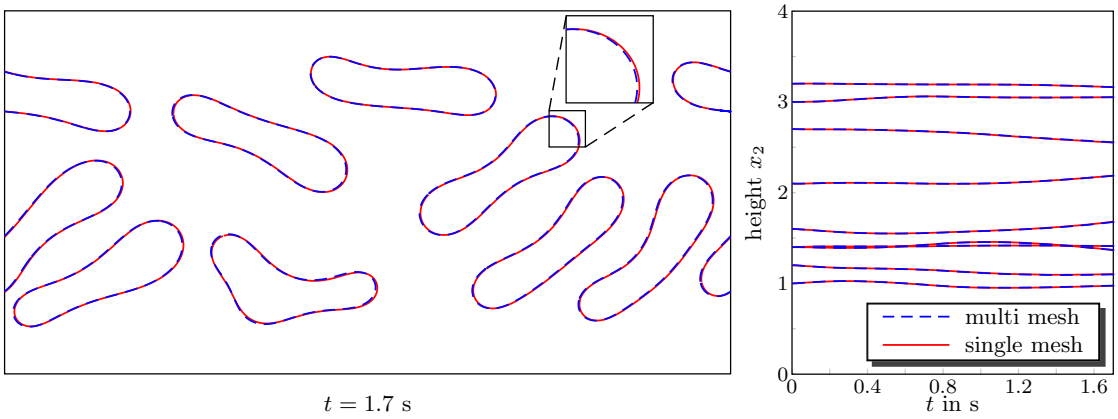
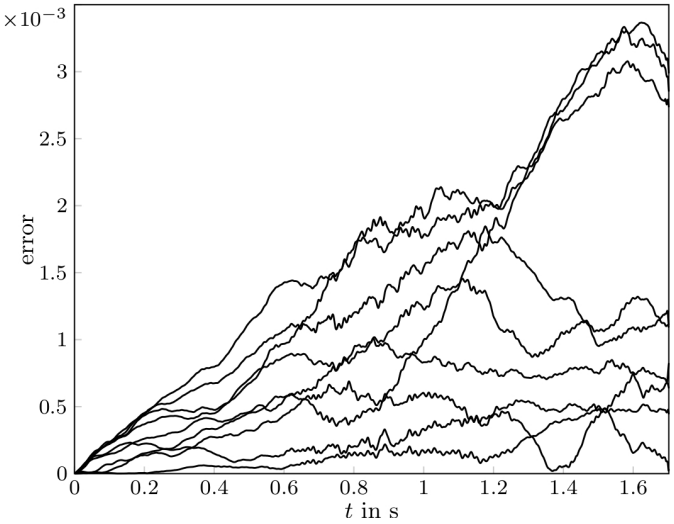


Figure 4.2: (left) Object shapes at final time, and (right) trajectories of center of mass position for each object. Results for multi-mesh and single-mesh computations are compared for 9 objects.

The trajectories of the objects, their center of mass points, are shown as well in comparison with the single mesh simulation results. By the end of the simulation, those cells have at least passed the tunnel more than two and a half round periodically. And both the object shapes and the trajectories of mass points are acceptable at that timestep. More precisely, Fig 4.3 shows the L2-error over time for each trajectory.



object i	$\|\|e\|\|_2$
1	0.000151
2	0.000300
3	0.001675
4	0.000218
5	0.000955
6	0.001123
7	0.000074
8	0.000690
9	0.000654

Figure 4.3: The point-wise error in the trajectories is measured over time for each object (left). L_2-error of the x_2 coordinate, i.e. $||e||_2 = (\sum_{n=1}^{NTS}(x^n_{2,SM} - x^n_{2,MM})^2/x^n_{2,SM})^{1/2}$ for each object, with SM - single mesh, MM - multi mesh and NTS - number of time steps (right).

To get exactly the same solution is impossible since we do assign a mesh with a coarser refinement set than before to each phase field variable, but we can see that there is only little difference in accuracy, which can be neglected. Then, the question is, whether the multi-mesh optimizes the performance. Table 4.1 lists the performance analysis. Here, we distinguish between two implementations, depending on whether we use the multi-mesh interpolation to solve Eq. (4.15) (denoted as "multi(variant)" in Table 4.1) or not (denoted as "multi(full)"). The results show a reduction of approximately 30% of the overall time cost for the lowest number of cells in the case of the full multi-mesh approach. And in the case of the variant implementation, we even get a better result. For the highest number of cells considered, the speedup is almost a factor of 6.

If we go into more detail, we find out that the main improvement comes from the phase field equation, which is in line with our expectations. In the single-mesh case, the assembling time increases a lot with the number of cells. But in the multi-mesh case, the time does not increase much, and it is much less when compared to the standard single-mesh finite element method. This is because now all the non-coupling terms in the phase

field equation is evaluated on a mesh which has much less refinement than before. Thus, the time cost for the calculation of the element matrix is also reduced. The situation for the solving time is similar. The time cost stays almost constant regardless of the number of cells, since now the number of DOFs on the phase field mesh is fixed. The main difference between the full multi-mesh approach and the variant implementation lies in the assembling time for the Navier-Stokes (NS) equation. In the case of the variant implementation, the summation of Eq. (4.15) is not done in the assembling step, but at the end of the previous time step and the situation goes back to the single-mesh case but without the coupling terms. Thus, the speedup in the assembling is perceivable. On the contrary, in the full multi-mesh approach, the assembling time for the NS equation is very heavy due to the overhead of the multi-mesh assembling algorithm and the fact that we did not reduce any DOFs on the fluid mesh. This can also be seen from the solution time of the NS equation. To conclude, the variant implementation is an alternative method to overcome the drawback of our multi-mesh method in this specific application. Note that even in the case of the full multi-mesh approach, the results are still much better compared to the standard method.

The accuracy and the overall speedup illustrate the applicability of the multi-mesh approach and its huge potential for other applications with multiple phase field variables.

4.1.5 Parallelization

Even if the multi-mesh method is used, for the highest number of objects we considered, the time cost for these objects to perform one round of the periodic movement is around 7 hours. We can further improve the performance if we can run the simulation in parallel. We have two ideas to achieve parallelization. First, we can use the parallel multi-mesh approach introduced in Chapter 3, adopting the domain decomposition strategy. Then, the comparison of the solution time does not make too much sense since we use a totally different kind of solver in parallel. Thus, the result is not given here. The second idea is to run the simulation on multiple threads using OpenMP. Inside each time iteration, we first run the Navier-Stokes problem in the main thread and then run each phase field problem simultaneously via an OpenMP loop parallelization. The overall time of the simulation under this strategy is shown in Table 4.2. The new results show a factor of 13 for the highest number of objects. To conclude, the multi-mesh method can be further combined with different kinds of parallel strategies, e.g. MPI, OpenMP, or even both, without too much effort. Most of the time this is the optimal solution we can achieve.

Table 4.1: Performance analysis of the single- and the multi-mesh strategy. The time for assembly and solve for the phase field (PF) equation is for one variable. The overall time accounts for this time times the number of objects. Times for mesh refinement and coarsening are also considered in others. All times are measured in sec and account for 100 time steps. The computation is done on TAURUS 2, the high performance computer at ZIH at TU Dresden.

	cells	all	assembly NS	solve NS	assembly PF $1 \dots n$	solve PF $1 \dots n$
single	5	1.641	101,0	66,1	114,4	152,7
multi(variant)	5	699	66,1	69,0	45,4	46,8
multi(full)	5	1.108	308,9	74,1	51,1	50,9
single	9	4.870	213,0	115,8	192,8	267,8
multi(variant)	9	1.274	106,4	112,7	50,8	43,7
multi(full)	9	2.517	865,5	130,1	60,8	51,0
single	13	9.141	334,9	154,4	245,4	364,6
multi(variant)	13	1.877	136,5	156,3	54,8	42,7
multi(full)	13	4.036	1.576,3	177,4	63,6	50,3
single	17	14.943	493,5	198,1	303,5	468,3
multi(variant)	17	2.559	166,3	200,6	61,2	42,1
multi(full)	17	6.594	2.558,7	225,6	69,2	48,7

Table 4.2: Overall time in sec for different number of objects with the combination of multi-mesh and OpenMP. The computation is done on TAURUS 2.

objects	all time
5	376
9	569
13	786
17	1.163

4.2 Dendritic growth

The numerical example we consider here is the dendritic growth, which describes the phenomenon that microstructures grow into multi-branching tree-like architectures during the process of solidification. In the following section, we introduce the model we use to solve the problem.

4.2.1 Governing equations

A powerful tool for describing the complex pattern evolution of the interface between mother and new phases in nonequilibrium state is the phase-field model. The model was originally proposed for simulating dendritic growth in undercooled pure melts and has also been extended to the solidification of alloys and so on [45]. For more reviews, we refer to [22, 7, 23, 36]. We adopted the model for quantitative simulations of the dendritic structures introduced by A.Karma and WJ.Rappel [23]. The model in non-dimensional form reads as follows:

$$A^2(n)\partial_t\phi = (\phi - \lambda u(1-\phi^2))(1-\phi^2) + \nabla\cdot(A^2(n)\nabla\phi) + \sum_{i=1}^{d}\partial_{x_i}\left(|\nabla\phi|^2 A(n)\frac{\partial_{A(n)}}{\partial_{x_i}\phi}\right) \tag{4.16a}$$

$$\partial_t u = D\nabla^2 u + \frac{1}{2}\partial_t\phi \tag{4.16b}$$

, where ϕ is the phase field variable and u is the thermal field variable. The parameter d in the phase field equation represents the dimension, which can be 2 or 3. The parameter D is the thermal diffusivity constant. $\lambda u(1-\phi^2))(1-\phi^2)$ is the non-linear coupling term between ϕ and u in the phase field equation, where the parameter $\lambda = \frac{D}{a_2}$, in which a_2 = 0.6267. Furthermore, function A is the anisotropy function and the form we use reads as:

$$A(n) = (1-3\epsilon)\left(1 + \frac{4\epsilon}{1-3\epsilon}\frac{\sum_{i=1}^{d}\phi_{x_i}^4}{|\nabla\phi|^4}\right) \tag{4.17}$$

, in which the coefficient ϵ controls the strength of the anisotropy and $n = \frac{\nabla\phi}{|\nabla\phi|}$ denotes the normal to the solid-liquid interface. In this setting, the phase-field variable is -1 in the liquid and 1 in the solid domain, and the melting temperature is set to be zero. We set $u = -\xi$ as a boundary condition to specify an undercooling. For the phase-field variable, we use zero-flux boundary conditions. The time integration is done using a semi-implicit Euler method, which yield a sequence of nonlinear stationary PDEs:

$$\frac{A^2(n_n)}{\tau}\phi_{n+1} + f + g - \nabla(A^2(n_n)\nabla\phi_{n+1}) - \mathcal{L}\left[A(n_n)\right] = \frac{A^2(n_n)}{\tau}\phi_n \tag{4.18a}$$

$$\frac{u_{n+1}}{\tau} - D\nabla^2 u_{n+1} - \frac{1}{2}\frac{\phi_{n+1}}{\tau} = \frac{u_n}{\tau} - \frac{1}{2}\frac{\phi_n}{\tau} \tag{4.18b}$$

with $f = \phi_{n+1}^3 - \phi_{n+1}$, $g = \lambda(1-\phi_{n+1}^2)^2 u_{n+1}$ and $\mathcal{L}[A(n_n)] = \sum_{i=1}^{d} \partial_{x_i} \left(|\nabla \phi_{n+1}|^2 A(n_n) \frac{\partial A(n_n)}{\partial_{x_i} \phi_n} \right)$
Then, we linearize the involved terms f and g with

$$
\begin{aligned}
f &\approx (3\phi_n^2 - 1)\phi_{n+1} - 2\phi_n^3 \\
g &\approx \lambda(1-\phi_n^2)^2 u_{n+1} \\
\mathcal{L} &\approx \sum_{i=1}^{d} \partial_{x_i} \left(|\nabla \phi_n|^2 A(n_n) \frac{\partial A(n_n)}{\partial_{x_i} \phi_n} \right)
\end{aligned}
$$

to obtain a linear system for ϕ_{n+1} and u_{n+1} to be solved at each time step. The discretization scheme we use is exactly the same as [53].

4.2.2 Adaptivity

The motivation of the usage of the multi-mesh approach in this problem is clear, due to the distinct solution behavior of the two components. The phase field variable ϕ needs a very high resolution along the solid-liquid interface in order to control the sensitive shape of the crystal, while in the region far away, it can be discretized with a very coarse mesh since the value is almost constant (1 for solid and -1 for liquid). On the contrary, the thermal field variable u has a relative smooth resolution everywhere in the domain, but the resolution along the interface is much lower compared to ϕ, see also Fig 1.3 or Fig 4.4. After the application of the multi-mesh method, we got the following coupling terms that defined on both of the meshes:

$$
\lambda(1-\phi_n^2)^2 u_{n+1} \quad \text{and} \quad -\frac{1}{2}\frac{\phi_{n+1}}{\tau}
$$

Since we adopt the full multi-mesh approach for these two terms, the multi-mesh assembling process inside AMDiS takes charge of the whole routine. There is no further modification necessary on the application-side. We only need to specify two refinement sets for the two components in the initial file. In order to prove that even under the parallel environment, the multi-mesh method is still favorable than the standard method, we tested the simulation with different number of processors. From a technical point of view, the parallel computation of the dendritic growth is very challenging. As the mesh changes in each iteration, the function checkMeshChange introduced in Section 3.3.2 must be called every time. The function itself includes: 1) the parallel mesh adaption to eliminate hanging nodes, 2) the update of all DOF-related data structures 3) the mesh repartitioning if necessary. Since the solid-liquid interface (the region with the most elements) changes and gets longer with the growth of the dendrite, even if at one timepoint, the global workload is well-balanced over all sub-domains, it becomes imbalanced rapidly in the next few steps. So the mesh repartitioning must be used in our case. All these sub-algorithms inside checkMeshChange, which now handle multiple meshes,

must show good performance so that the efficiency of the parallel codes is comparable, or even better than the efficiency of the sequential ones.

4.2.3 Performance

We considered a 2D dendrite using linear finite elements with the parameters: $D = 1.0$, $\epsilon = 0.05$. For the mesh adaption, the equidistribution mark strategy with $\theta_R = 0.8$ and $\theta_C = 0.2$ was used. The tolerance for the phase field was set to $tol_\phi = 0.5$ and the tolerance for the thermal field was $tol_u = 0.25$. For the error estimator, only the jump residuum was used. To speedup the computation, we have employed the symmetry of the solution and limited the computation to the upper right quadrant with a domain size of 1000 in each direction. Furthermore, we started the simulation from the initial time 0 up to the end time $1.3 \cdot 10^4$ with the constant timestep $\tau = 1.0$. Because of the mesh adaption, we increased the number of processors during the whole simulation, see Fig 4.4. The initial radius of the round solid field is set to 3, which is not evident in the scale of the whole domain.

time	nproc
0,0 - 800.0	2
800,0 - 2500,0	4
2.500,0 - 4.000,0	8
4.000,0 - 7.000,0	16
7.000,0 - 10.000,0	32
10.000,0 - 13.000,0	64

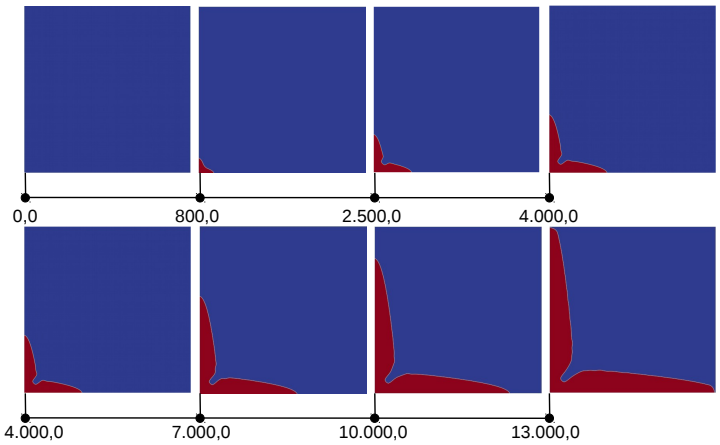


Figure 4.4: (left) Number of processors used over time and (right) the corresponding simulation results at given timepoints

The recovery of the simulation is based on the ARH file introduced in Section 1.2.2.3. We are also interested in the parallelization of the dendrite. Some related information is given in Fig 4.5. On the left side, the mesh comparison between the phase field and the thermal field at given timepoint is shown. The refinement structure of the mesh is denoted by blue lines, and the partition of the mesh in each sub-domain is marked by a different color. We can see that at this timepoint, both meshes are well-distributed. Among all the partitions, the green one, which does not contain heavy refinement at all, has the largest area, while the other partitions, which cover the interface region, are

much smaller. The good load-balancing is also visible by the low value of the imbalancing factor near the first point, shown on the right side. We picked up the imbalancing factors every 10 time steps. From the figure, we can see that during the following period of time, we are able to control the workload in the multi-mesh case in the same extent as the workload in the single-mesh case. The significant drop between neighbor points indicates that a repartitioning took place in between.

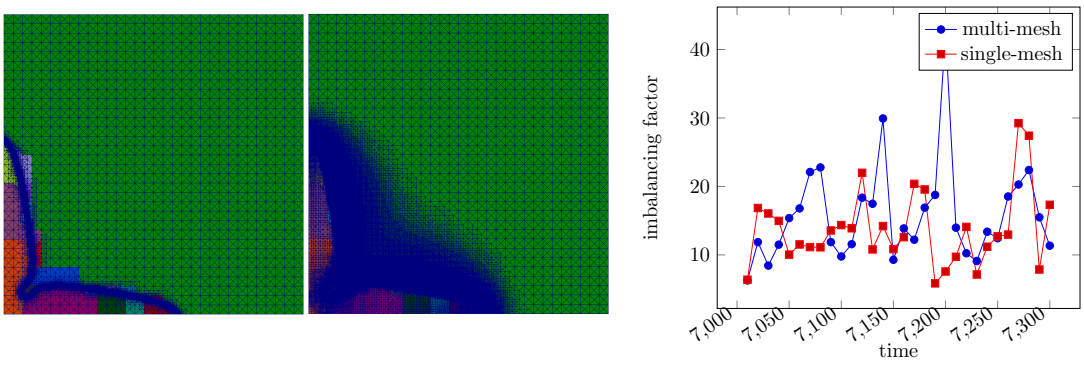


Figure 4.5: (left) Mesh comparison between the phase field and the thermal field at timepoint $7.000,0$ in the multi-mesh case, and (right) imbalancing factors of the single- and the multi-mesh methods starting from timepoint $7.000,0$

We now consider the simulation in terms of performance. Fig 4.6 shows the complete time comparison between the parallel single- and multi-mesh approaches. First of all, we have a look at the overall speedup. The speedup begins with a negative value, i.e. the single-mesh routine runs faster in the first time period. This is due to the fact that the interface region of the phase field is too small at the beginning of the simulation, then the difference between the number of DOFs on the phase field and the thermal field is inconspicuous. As a result, the advantage of the multi-mesh method disappears. But, with the growth of the dendrite, the overall speedup rises and keeps at least 30% until the end of the simulation. From the second graph, we can see that the time for the assembling is always more or less equal regardless of the strategy we choose. This is because most of the non-coupling terms in Eq. (4.18) are located on the phase field equation, and the the phase field meshes between the single- and the multi-mesh method are quite similar (see the number of DOFs in Table 4.3). And, we need to perform the additional work for the coupling terms. On the contrary, the solver and the error estimator are largely benefited from the saving of DOFs, especially the solver, which enjoys a speedup up to 55% in performance. On the other hand, in line with our expectations, the multi-mesh parallel mesh adaption does cost more time than before, but not much. And, if we compare the time for the parallel mesh adaption with the time for the assembling, the solver, and even the error estimator, we find that it can

be totally neglected. So, the result is acceptable for us. Furthermore, the time for the mesh repartitioning is also neglectable small due to our strategy that the function is only triggered every 20 iterations. To sum up, we proved that the multi-mesh method is superior to the standard method even in the parallel environment. Our method not only increases performance, but also reduces the memory usage tremendously, which is always one of the bottlenecks in the simulation of large-scale problems. This is explained in more detail in Table 4.3, where information on the global matrix of the linear system at the last timepoint is shown.

	single mesh	multi mesh
DOFs	2.014.350 × 2	1.697.659 + 157.550
Matrix none zero values	56.137.808	25.332.378
Matrix size	7.65e+02 MB	3.47e+02 MB
BiCGStab iterations	119	87

Table 4.3: Information on the matrix of the linear system at the last timepoint

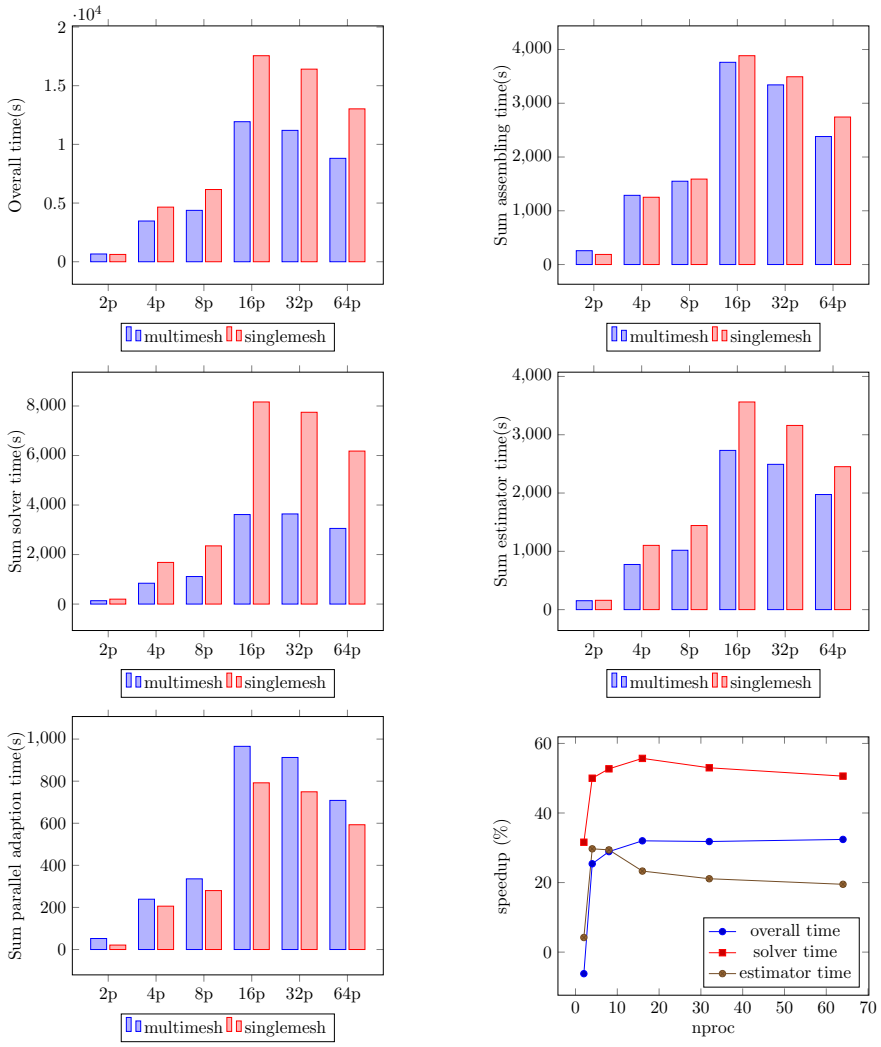


Figure 4.6: The first five graphs are the time comparison of overall runtime, assembling, solver, estimator, parallel adaption between the single- and the multi-mesh methods. The last graph is the time speedup resulting from the multi-mesh method. The computation is done on TAURUS 2.

CHAPTER 5

Conclusion and outlook

In the first part of this thesis, we introduced the basic concepts and an advanced numerical tool, the multi-mesh method, implemented in the AMDiS environment. Basically speaking, the method uses independently adaptive meshes for different variables in the systems of nonlinear, time-dependent PDEs in order to improve the efficiency of adaptive finite element simulations. We pointed out that the current implementation of the multi-mesh method is not complete. Then, the second part of the thesis filled the absent parts. First of all, we generalized the number of the meshes used in the multi-mesh method from two to an arbitrary number, allowing the method to be applied to those applications, where more than two variables are involved. The generalization includes the extension of the mesh traversal algorithm and also of the application of the transformation matrix. Besides that, we applied the concept of the transformation matrix further to the situation, where a bunch of coefficient function spaces is attached to the operator terms inside PDEs. Data structures and algorithms which allow to implement an efficient generalized multi-mesh approach are introduced after the theory. Furthermore, we pointed out that it would be a pity if this method was not available in parallel since users are accustomed to improving performance by exploiting parallel resources rather than using the multi-mesh strategy on one single processor. We discussed the possibility of the combination of the multi-mesh method and the parallelization, and we found out that the difficulty lies not in the method itself, but in the parallel management of multiple meshes, including the enumeration of the DOFs defined on them. So, we temporarily shifted our topic from the multi-mesh to our parallelization approach. We presented the implementation of adaptively refined and distributed mesh data structures based on mesh structure codes, and the parallel linear solution methods based on DOF-related information containers. We also showed a very important parallel algorithm, the parallel mesh adaption, in detail. In the last part of the thesis, we illustrated two numerical

experiments of the presented multi-mesh method to prove that our method is superior in contrast to the standard single-mesh finite element method in terms of performance. In the first example, we considered the multi-phase flow problem with a hydrodynamic phase field model. In the second example, we considered the dendritic growth problem with a phase field model. Both numerical results were shown with detailed analyses. The multi-mesh method we presented is implemented on top of the existing codes in the AMDiS environment, but it does not rely on any special data structure. So, in principle, the main idea and the software concepts are also portable to any other finite element toolbox.

We have several ideas to improve our work in the future, most of which are related to the numerical applications. Although we already presented two problems in this thesis, none of them includes the situation, where the coefficient function spaces live on an independently refined mesh. We proved the correctness of the implementation of the coefficient function spaces using simple examples, but so far we didn't find a real application, in which our approach shines. This can be one future task. Moreover, we are lack of a 3D example. We tried to run the dendritic growth in 3D, but the solution time is negligible compared to the assembling time. Thus, we didn't obtain any benefit from the multi-mesh approach. Fixing this problem can be another future task. Further examples, that might enjoy large computational savings due to the usage of the multi-mesh method, are also welcome, e.g. optimal control problems, diffuse interface and diffuse domain approximations for PDEs to be solved on surfaces that are within complicated domains, etc.

Bibliography

[1] S. Aland, S. Egerer, J. Lowengrub, and A. Voigt. Diffuse interface models of locally inextensible vesicles in a viscous fluid. *Journal of Computational Physics*, 277:32–47, 2014.

[2] Ivo Babuška and Werner C Rheinboldt. A-posteriori error estimates for the finite element method. *International Journal for Numerical Methods in Engineering*, 12(10):1597–1615, 1978.

[3] S Balay, S Abhyankar, M Adams, J Brown, P Brune, K Buschelman, V Eijkhout, W Gropp, D Kaushik, M Knepley, et al. Petsc users manual revision 3.5. *Argonne National Laboratory (ANL)*, 2014.

[4] Satish Balay, Kris Buschelman, William D Gropp, Dinesh Kaushik, Matthew G Knepley, L Curfman McInnes, Barry F Smith, and Hong Zhang. Petsc. *See http://www. mcs. anl. gov/petsc*, 2001.

[5] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry F Smith. Petsc, the portable, extensible toolkit for scientific computation. *Argonne National Laboratory*, 2:17, 1998.

[6] Michel Bercovier and Olivier Pironneau. Error estimates for finite element method solution of the stokes problem in the primitive variables. *Numerische Mathematik*, 33(2):211–224, 1979.

[7] William J Boettinger, James A Warren, Christoph Beckermann, and Alain Karma. Phase-field simulation of solidification. *Materials Research*, 32(1):163, 2002.

[8] Umit V Catalyurek, Erik G Boman, Karen D Devine, Doruk Bozdağ, Robert T Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, 2009.

[9] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–96, 2002.

[10] Yana Di and Ruo Li. Computation of dendritic growth with level set model using a multi-mesh adaptive finite element method. *Journal of Scientific Computing*, 39(3):441–453, 2009.

[11] Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation*, volume 144. SIAM, 2015.

[12] Maksymilian Dryja, Barry F Smith, and Olof B Widlund. Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. *SIAM journal on numerical analysis*, 31(6):1662–1694, 1994.

[13] Maksymilian Dryja and Olof B Widlund. A feti-dp method for a mortar discretization of elliptic problems. In *Recent Developments in Domain Decomposition Methods*, pages 41–52. Springer, 2002.

[14] Q. Du, C. Liu, R. Ryham, and X. Wang. A phase field formulation of the Willmore problem. *Nonlinearity*, 18:1249–1267, 2005.

[15] Q. Du, C. Liu, and X. Wang. Simulating the deformation of vesicle membranes under elastic bending energy in three dimensions. *Journal of Computational Physics*, 212:757–777, 2006.

[16] Kenneth Eriksson and Claes Johnson. Adaptive finite element methods for parabolic problems i: A linear model problem. *SIAM Journal on Numerical Analysis*, 28(1):43–77, 1991.

[17] Charbel Farhat, Michel Lesoinne, Patrick LeTallec, Kendall Pierson, and Daniel Rixen. Feti-dp: a dual–primal unified feti method—part i: A faster alternative to the two-level feti method. *International journal for numerical methods in engineering*, 50(7):1523–1544, 2001.

[18] Peter Gottschling and Andrew Lumsdaine. The matrix template library 4, 2009.

[19] Peter Gottschling, David S Wise, and Michael D Adams. Representation-transparent matrix algorithms with scalable performance. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125. ACM, 2007.

[20] Thomas Grätsch and Klaus-Jürgen Bathe. A posteriori error estimation techniques in practical finite element analysis. *Computers & structures*, 83(4):235–265, 2005.

[21] Xianliang Hu, Ruo Li, and Tao Tang. A multi-mesh adaptive finite element approximation to phase field models. *Communications in Computational Physics*, 5(5):1012–1029, 2009.

[22] Alain Karma and Wouter-Jan Rappel. Phase-field method for computationally efficient modeling of solidification with arbitrary interface kinetics. *Physical Review E*, 53(4):R3017, 1996.

[23] Alain Karma and Wouter-Jan Rappel. Quantitative phase-field modeling of dendritic growth in two and three dimensions. *Physical review E*, 57(4):4323, 1998.

[24] George Karypis. Metis and parmetis. In *Encyclopedia of Parallel Computing*, pages 1117–1124. Springer, 2011.

[25] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[26] Axel Klawonn, Olof B Widlund, and Maksymilian Dryja. Dual-primal feti methods for three-dimensional elliptic problems with heterogeneous coefficients. *SIAM Journal on Numerical Analysis*, 40(1):159–179, 2002.

[27] Damien Lebrun-Grandié, Jean Ragusa, Bruno Turcksin, and Pavel Solin. Adaptive multimesh hp-fem for a coupled neutronics and nonlinear heat conduction problem. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering. American Nuclear Society*, pages 8–12, 2011.

[28] Ruo Li. On multi-mesh h-adaptive methods. *Journal of Scientific Computing*, 24(3):321–341, 2005.

[29] Siqi Ling, Wieland Marth, Simon Praetorius, and Axel Voigt. An adaptive finite element multi-mesh approach for interacting deformable objects in flow. *Computational Methods in Applied Mathematics*, 2016.

[30] Jan Mandel. Balancing domain decomposition. *Communications in numerical methods in engineering*, 9(3):233–241, 1993.

[31] Jan Mandel and Marian Brezina. Balancing domain decomposition for problems with large jumps in coefficients. *Mathematics of Computation of the American Mathematical Society*, 65(216):1387–1401, 1996.

[32] Jan Mandel and Bedřich Sousedík. Bddc and feti-dp under minimalist assumptions. *Computing*, 81(4):269–280, 2007.

[33] Wieland Marth, Sebastian Aland, and Axel Voigt. Margination of white blood cells: a computational approach by a hydrodynamic phase field model. *Journal of Fluid Mechanics*, 790:389–406, 2016.

[34] JL Meek. A brief history of the beginning of the finite element method. *International journal for numerical methods in engineering*, 39:3761–3774, 1996.

[35] Ricardo H Nochetto, Kunibert G Siebert, and Andreas Veeser. Theory of adaptive finite element methods: an introduction. In *Multiscale, nonlinear and adaptive approximation*, pages 409–542. Springer, 2009.

[36] Wouter-Jan Rappel and Alain Karma. Quantitative phase-field modeling of dendritic growth in two and three dimensions. In *APS March Meeting Abstracts*, volume 1, page 1801, 1996.

[37] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013.

[38] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[39] Alfred Schmidt. A multi-mesh finite element method for phase-field simulations. *Lecture Notes in Computational Science and Engineering*, 32:208–217, 2003.

[40] Alfred Schmidt and Kunibert G Siebert. *ALBERT: An adaptive hierarchical finite element toolbox*. Albert-Ludwigs-Univ., Math. Fak., 2000.

[41] Marc Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[42] P Solin, J Cerveny, L Dubcova, and D Andrs. Adaptive multi-mesh hp-fem for linear thermoelasticity problems. *J. Comput. Appl. Math.(accepted)*.

[43] Pavel Solín, J Cerveny, Lenka Dubcova, and David Andrs. Monolithic discretization of linear thermoelasticity problems via adaptive multimesh hp-fem. *Journal of computational and applied mathematics*, 234(7):2350–2357, 2010.

[44] Pavel Solin, Lenka Dubcova, and Jaroslav Kruis. Adaptive hp-fem with dynamical meshes for transient heat and moisture transfer problems. *Journal of computational and applied mathematics*, 233(12):3103–3112, 2010.

[45] Toshio Suzuki, Machiko Ode, Seong Gyoon Kim, and Won Tae Kim. Phase-field model of dendritic growth. *Journal of Crystal Growth*, 237:125–131, 2002.

[46] Andrea Toselli and Olof B Widlund. *Domain decomposition methods: algorithms and theory*, volume 34. Springer, 2005.

[47] Rüdiger Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. *Journal of Computational and Applied Mathematics*, 50(1):67–83, 1994.

[48] Simon Vey. Adaptive finite elements for systems of pdes: Software concepts, multilevel techniques and parallelization. 2007.

[49] Simon Vey and Thomas Witkowski. *AMDiS tutorial*. TU Dresden., Math. Fak., 2012.

[50] Axel Voigt and Thomas Witkowski. Hybrid parallelization of an adaptive finite element code. *Kybernetika*, 46(2):316–327, 2010.

[51] Axel Voigt and Thomas Witkowski. A multi-mesh finite element method for lagrange elements of arbitrary degree. *Journal of Computational Science*, 3(5):420–428, 2012.

[52] T Witkowski, S Ling, S Praetorius, and A Voigt. Software concepts and numerical algorithms for a scalable adaptive parallel finite element method. *Advances in computational mathematics*, 41(6):1145–1177, 2015.

[53] Thomas Witkowski. Software concepts and algorithms for an efficient and scalable parallel finite element method. 2013.

[54] OC Zienkiewicz. Origins, milestones and directions of the finite element method—a personal view. *Archives of Computational Methods in Engineering*, 2(1):1–48, 1995.

Selbstständigkeitserklärung

Die eingereichte Dissertation zum Thema:

"Solving multi-physics problems using adaptive finite elements with independently refined meshes"

wurde am Institut für Wissenschaftliches Rechnen der Technischen Universität Dresden unter Betreuung durch Prof. Dr. rer. nat. Axel Voigt angefertigt. Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Dresden, 19.10.2016
Siqi Ling